Rapport de Stage - Cheddar
Licence Informatique

Philippe Normand
Sebastien Herry
Regis Prevot
Wanig Guillo
Gwendal Oliva

26 juillet 2002

Table des matières

Table des figures

Introduction

Dans le cadre de la Licence Informatique, il nous a été demandé d'effectuer un stage de fin d'année d'une durée de deux mois. Ce stage a pour but d'améliorer nos connaissances et surtout de nous permettre de mettre en pratique l'enseignement de licence.

Nous avons effectue ce stage à l'Université de Bretagne Occidentale dans le département informatique sous la direction de Franck Singhoff professeur dans ce département et chercheur au laboratoire LIMI.

Ce stage consiste a poursuivre le développement d'un outils d'ordonnancement appelé Cheddar. Celui-ci a commencé a être développés par différents groupes de projets d'iup ,par Mr Singhoff et Mr Legrand thésard. Cet outil est utilisé dans l'enseignement au département informatique. Il permet de simuler l'ordonnancement de taches sur un ou plusieurs processeurs avec plusieurs protocoles d'ordonnancement tel que RM et DM.

Il nous a été demandé de poursuivre le développement de cette application tous en essayant d'améliorer les fonctionnalités déjà existantes. Les principaux points sont l'ajout des précedances et d'un canevas permettant de les gérer, ainsi que la mise en place du concept de ressources et de partage de celle-ci. Pour permettre un développement de meilleure qualité il a été décidé de créer deux groupes de développements, un pour les précédences et le canevas l'autre pour les ressources.

D'une part nous présenterons l'Université de bretagne occidentale, ainsi que le département informatique et le laboratoire LIMI ou nous avons effectué notre stage. D'autre part nous décrirons le projet cheddar, les outils utilisé durant ce stage, et l'état d'avancement du projet lors de notre arrivée. Enfin, nous exposerons la tache réalisé lors de ces deux mois qui comprend la mise en place des contraintes de précédences et la création des ressources partagées.

Chapitre 1

Présentation de l' UBO

1.1 Historique

L'Universite de Bretagne Occidentale est constitue des UFR suivantes :

- Lettres et Sciences Sociales
- Droit, économie et gestion
- Médecine
- Odontologie
- Sciences et Techniques
- Sport et Éducation Physique

Mais egalement d'institut d'enseignement et d'ecoles (Euro-Institut d'Actuariat (EURIA),l'Institut d'Administration des Entreprises (IAE), IUP Ingénierie Informatique, l'IUT de Brest et de Quimper ...) , d'instituts de recherche, et de services communs (bibliothèques universitaires,...).

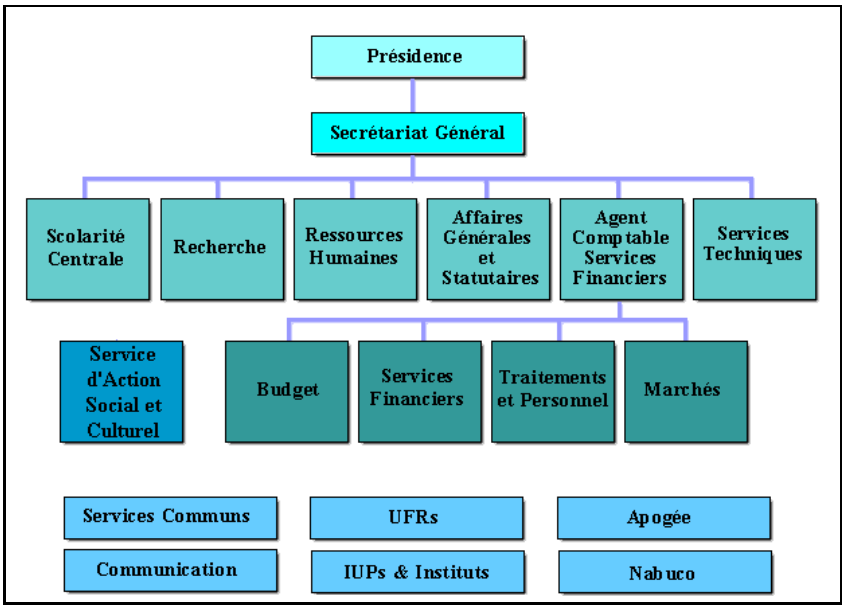


FIG. 1.1 – Organigramme de l'UBO

L'UBO acceuille environ 18000 etudiants sur 3 sites differents (Brest , Morlaix, Quimper), elle reunie egalement 840 enseignants et chercheurs repartis en une cinquantaines de groupes de

recherche.

1.2 Le département Informatique

1.2.1 L'UFR sciences et techniques

UFR Sciences et Techniques propose un large éventail de formations et de cursus complets allant du DEUG jusqu'au DESS, au DEA et au Doctorat. Outre le choix entre de nombreuses disciplines (Biologie, Chimie, Electronique, Géologie, Informatique, Mathématiques, Physique, Urbanisme), chaque étudiant peut adapter son parcours et s'orienter tout au long de sa formation. Gage de qualité, la plupart de ces formations s'appuient sur des laboratoires de recherche de haut niveau, reconnus tant sur le plan national qu'international.

Ces formations sont une alternative aux préparations aux grandes écoles. Un nombre croissant d'étudiants passent en effet par l'UFR Sciences avant d'intégrer les Ecoles d'Ingénieurs, et valorisent par la suite leur parcours universitaire original et formateur.

l'UBO et l'UFR Sciences et Techniques se tourne résolument vers l'avenir avec et pour ses étudiants : nouvelles technologies, renouvellement pédagogique, formations professionnelles, développement des relations internationales.

1.2.2 Le département informatique

Le Département d'Informatique est un département d'enseignement et de recherche. Le Département est responsable des enseignements et gère les équipements relevant de ses activités pédagogiques.

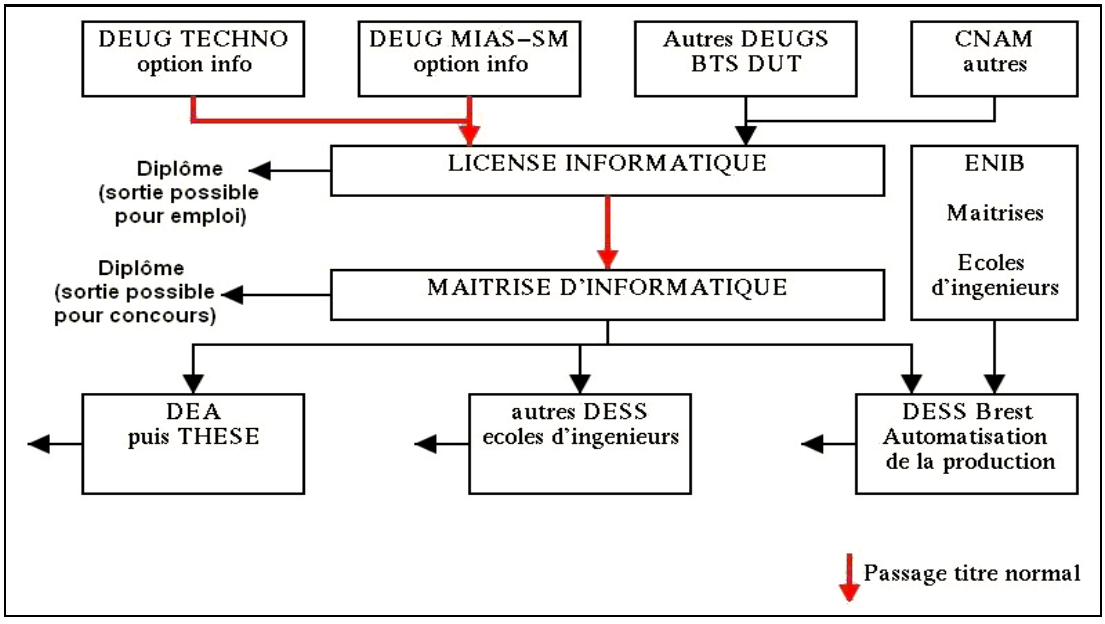


FIG. 1.2 – Cursus au département informatique

Il est administré par un conseil de département élu constitué de :

- L'ensemble des professeurs
- Autant d'autres enseignants qu'il y a de professeurs
- Autant d'autres enseignants qu'il y a de professeurs

- Un représentant du personnel technique
- Un étudiant de deuxième cycle et un étudiant de troisième cycle

Le conseil de département élit, parmi ses membres enseignants, un responsable et un responsable adjoint.La durée des divers mandats est de deux ans. Le responsable actuel est Lionel Marcé.

L'Equipe pédagogique

- 2 professeurs
- 16 maîtres de conférences
- Un assistant
- Un PAST
- Un MCF associé
- 1 professeur agrégé
- 1 attaché temporaires d'enseignement et de recherche
- Ces titulaires sont assistés par de nombreux vacataires notamment pour le monitorat en premier cycle et
- Le tutorat

1.3 Le Laboratoire LIMI

L' EA 2215, unité INFORMATIQUE a été créée, il y a 8 ans, à l'Université de Bretagne Occidentale. Elle regroupe principalement les 25 enseignants-chercheurs en informatique (27ème) enseignant dans le département d'informatique de l'UFR Sciences et à l'IUP d'ingénierie informatique, et la dizaine d'enseignants en informatique et génie informatique (27ème et 61ème) enseignant à l'ENIB, auxquels se rajoutent quelques enseignants dispersés sur l'Université en particulier à l'IUT.

L'unité est dirigée par le Prof. L. Marcé assisté d'un conseil d'unité comprenant les MCF suivants : M. P. Le Parc, Melle S. Gire, M. D. Sarni, M. B. Pottier, M. V. Ribaud , M. V. Rodin, M. J. Tisseau.

L'unité comprend 4 équipes :

- Architecture et Système (A&S) animée par B. Pottier, comprenant deux thématiques appliquées en
- génie logiciel et en architecture de machines .
- Informatique Fondamentale (In-Fo), animée par D. Sarni. Ce projet comprend des théoriciens .
- Langages et Interfaces pour Machines Intelligentes ou non (LIMI), animée par L. Marcé,.
- Réalité virtuelle (AREVI), animée par J. Tisseau.
- Les deux parties de l'équipe, pour l'U.B.O (projets AS, In-Fo, LIMI) et pour l'ENIB (AREVI) ont actuellement des politiques et des financements indépendants.
- Elles organisent néanmoins en commun depuis 1996, annuellement une journée de séminaires pour faire le point sur les avancées de l'année.

Le projet Langages et Interfaces pour Machines Intelligentes (LIMI) est né en 1990, suite à la nomination à Brest de L. Marcé comme Professeur d'informatique. Avant la création de l'EA 2215 le LIMI regroupait la communauté universitaire recherche en Génie Informatique localisée à Brest, ce qui lui a permis de se structurer en projets. Suite à l'organisation de l'EA 2215 en projets, le laboratoire LIMI s'est réparti en plusieurs projets dont le projet LIMI ici décrit.

La problématique générale est l'étude des langages, interfaces et environnements informatiques pour la supervision et l'exploitation des machines intelligentes ou non, plus généralement des systèmes évoluant avec le temps, avec comme applications les langages de spécification et de contrôle

d'exécution pour le temps réel, les méthodes et les langages de spécification pour la productique, la téléproductique, les architectures logicielles et matérielles pour la robotique, la télérobotique, les modèles et architectures pour les simulateurs de processus biologiques, pour le test de cartes hybrides, pour le dépannage. Le LIMI a développé une expertise autour de la modélisation des langages de supervision des systèmes réactifs et de la vérification de propriétés temporisées pour ces langages, basées sur les langages synchrones et les automates temporisés.

Le LIMI est également expert dans les environnements de programmation pour la robotique mobile et la productique.

La recherche liée au LIMI est en partie enseignée dans l'option de la maîtrise informatique systèmes réactifs et intelligents, ainsi que dans le module systèmes réactifs du DEA sciences et technologies de télécommunications.

Chapitre 2

Le Projet Cheddar

CHEDDAR est un simulateur d'ordonnancement temps réel à vocation éducatif, développé en Ada95 et en GtkAda pour l'interface graphique sous licence GPL. Cheddar peut être exécuté sous différents Systèmes d'exploitation (SunOS, GNU/Linux, MS Windows).

Les objectifs de Cheddar et le travail à faire sur celui-ci sont présentés par la suite.

2.1 Objectifs

2.1.1 Ordonnancement temps réel

Principe de l'ordonnancement temps réel
L'ordonnancement utilise des Algorithmes qui vont permettre de planifier l'exécution des tâches qui pourront répondre différemment suivant le système utilisé. Les Algorithmes d'ordonnancement ont aussi pour but de permettre aux tâches de respecter leur échéance, une échéance étant la durée á laquelle la tâche doit avoir terminée.
Il existe deux catégories de tâche :

- Les tâches périodiques qui seront le plus souvent utilisée pour le contrôle. L'échéance doit être de durée inférieure ou égale à la période.
- Les tâches apériodiques qui n'ont pas de déclenchement prévisible et qui peuvent être activées par des tâches périodique en cas d'urgence.

Les tâches peuvent aussi être préemptible ou non préemptible, dans le cas d'une tâche préemptible (figure 2.1) celle pourra être interrompu par le système et mise en attente pour permettre á une tâche de plus forte priorité d'avoir accès au processeur, tandis que dans le cas d'une tâche non préemptible (figure 2.2) une tâche ayant une priorité plus haute devra attendre la fin de l'exécution de la tâche en cours pour pouvoir accès au processeur.

C : Capacité est la durée de la tâche.
P : Est la priorité de la tâche, plus elle est base plus la priorité est haute.
ST : Représente son temps de départ.

Chaque tâche doit s'exécuter intégralement dans un intervalle de temps fixe, qui dépend de la périodicité de la tâche, des contraintes, des caractéristiques du système, et sera caractérisée temporellement par quatre valeurs numériques (période, échéance, Date d'activation et la capacité). Pour les ordonnancements à priorités fixe il existe des algorithmes d'ordonnancement optimaux

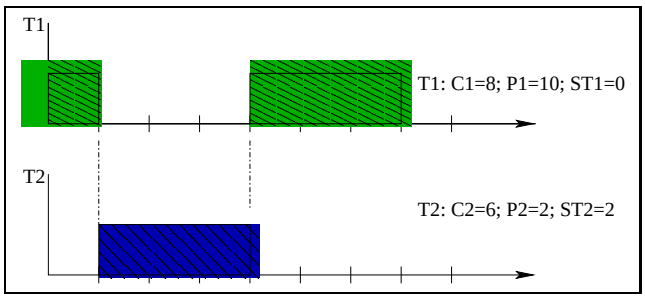


FIG. 2.1 – tâches préemptible

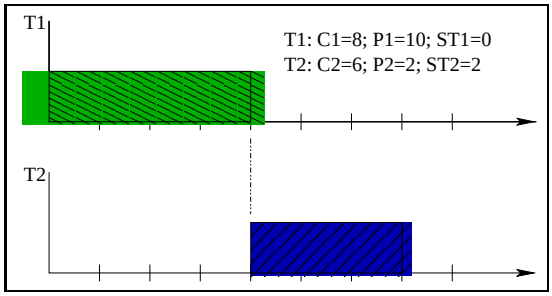


FIG. 2.2 – tâches non préemptible

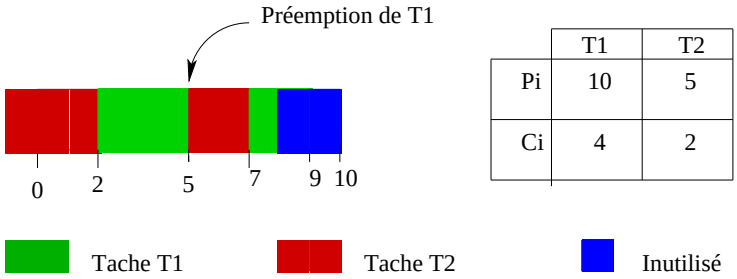
pour les systèmes de tâches indépendantes, mais un systèmes temps réel doit pouvoir accéder á des ressources qui doivent être partagées, ce qui peut entraîner des problèmes dans le cas d'un système avec des tâches préemptive qui induisent que les résultats d'optimalité ne sont plus valide, d'autres algorithmes sont donc nécessaire et sont présentés dans la partie 3.

Les ordonnancements à priorités dynamique, se rapproche d'un système temps partagés comme par exemple Unix, ainsi les tâches ont leur priorités modifiés lors de l'exécution, de cette façon une tâche à peu de chance de se retrouver bloquées.

Présentation des algorithmes à priorité fixe

L'algorithme Rate Monotonic (RM)

L'algorithme Rate Monotonic s'applique aux tâches périodiques seulement, les priorités sont fixés au départ en la mettant égale à l'inverse de la périodicité, la plus forte priorité est élue.



	T1	T2
Pi	10	5
Ci	4	2

FIG. 2.3 – Exemple de l'algorithme RM

Dans l'exemple de la figure 2.3, la tâche T2 est plus prioritaire car sa période est plus petite que la tâche T1, la tâche T1 commence son exécution quand T2 à fini mais ne peut pas finir son

exécution car T2 est plus prioritaire. La tâche T1 est donc préempté et pourras finir son exécution lorsqu'aucune tâche plus prioritaire ne voudras s'exécuter.

L'algorithme Deadline Monotonic (DM)
L'algorithme Deadline Monotonic calcule la priorité par rapport aux délais critiques, plus le délai critique sera petit, plus la tâche sera prioritaire.

Algorithmes à priorité dynamique

On retrouve aussi dans Cheddar des algorithmes à priorité dynamique tel que EDF(Earliest Deadline First), LLF(Least Laxity First), HPF(Hightiest Priority First).

2.1.2 Résultat obtenu avec Cheddar

La figure 2.4 contient trois tâches périodiques et préemptible, ordonnancé avec l'algorithme RM, on peut voir que T2 est préempté 13 fois, pour permettre l'exécution de T1 qui à une priorité plus basse et est donc plus prioritaire. De plus T2 se trouve être bloqués à cause d'une ressource PCP [1], qui est utilisée par la tâche T1, alors que T2 veut y accéder.

2.1.3 Les contraintes de précédence

Un autre objectif du stage est de réaliser une vue pour Cheddar. Cette vue permet de dessiner et de visualiser les objets (tâches, tampons, messages) de Cheddar. Elle doit être indépendante de Cheddar du point de vue de son utilisation (création/destruction d'items). Elle doit de plus mettre en évidence les contraintes de précédence que l'utilisateur peut mettre en place dans Cheddar.

[1] Priority Ceiling protocol

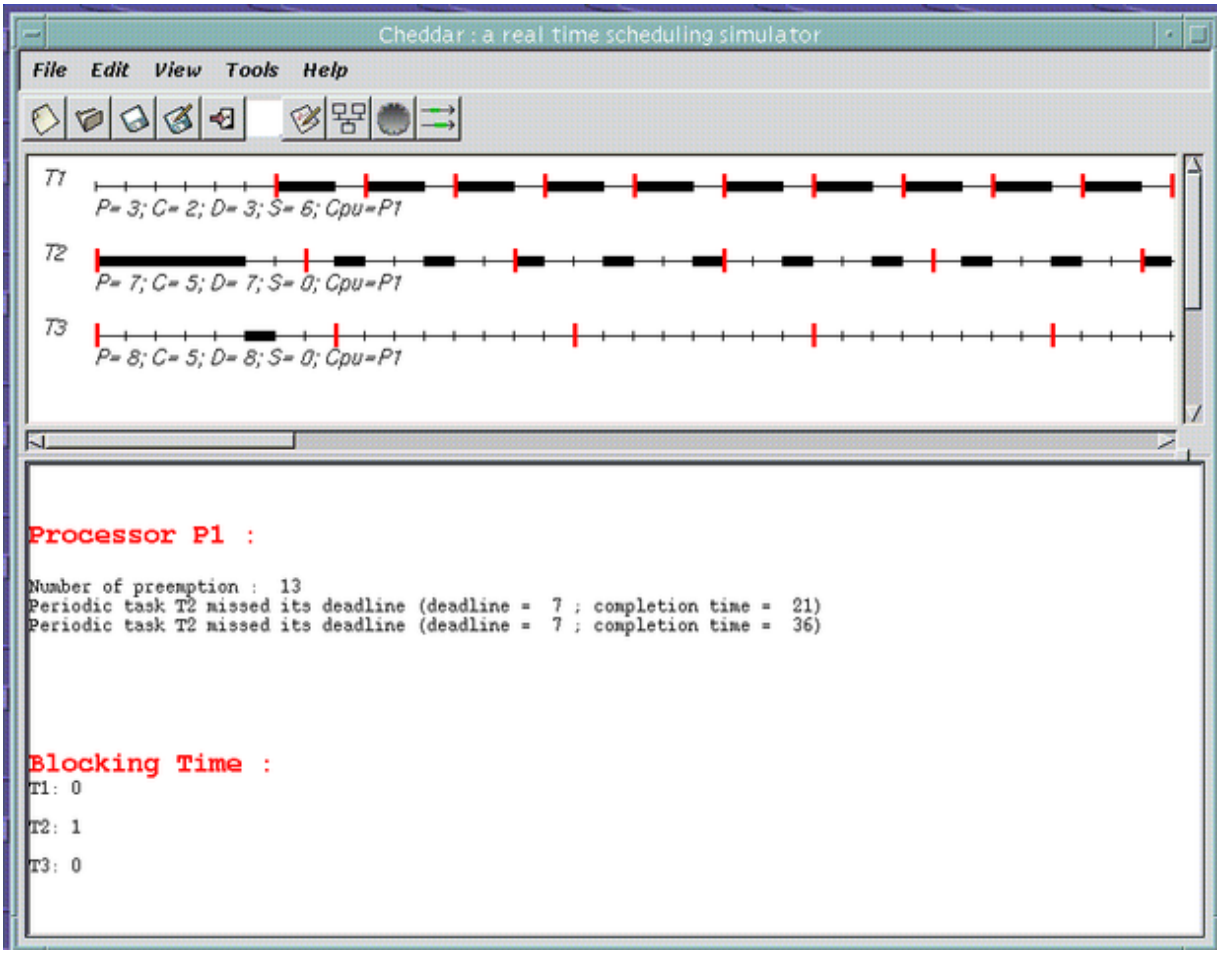


FIG. 2.4 – capture de Cheddar

2.2 Outils

Cheddar est développé en GtkAda qui est le binding Ada de Gtk+ [2]. Gtk+ est un toolkit graphique qui a été initialement écrit pour 'The Gimp' [3], un logiciel de dessin fonctionnant sous GNU/Linux.

2.2.1 Le Langage Ada95

Ada est un langage qui ressemble a priori à du Pascal, et c'est normal puisqu'il a été créé en se basant sur celui-ci. Le langage fut baptisé Ada en l'honneur de Lady Ada Lovelace qui fut la fille de Lord Byron.

Le langage est caractérisé par son typage fort, son traitement de4s exceptions, un abstraction des données, une interface avec des programmes écrits dans d'autres langages,... De plus, il ne respecte pas la casse (différenciation minuscules/majuscules).

Structure d'un programme

Le bout de code ci-dessous, un fois compilé avec le compilateur Gnat, affiche une chaîne de caractères sur la sortie standard.

Le fichier contient une seule procédure qui sera exécutée.

La procédure `Put_Line` fait partie du sous-package `Text_IO` de `Ada`. Donc pour utiliser cette procédure, on aurait pu l'appeler par `Ada.Text_IO.Put_Line(''Ma chaine'')`, mais en pratique, on peut utiliser toutes les fonctions et procédures d'un package en l'incluant au début du fichier source avec les clauses `with` et `use`

Les Packages

Les packages permettent de regrouper un ensemble de déclarations, de procédures et de fonctions. Un package est constitué de :

1. La spécification qui se trouve dans un fichier avec extension `.ads` se décompose en deux parties :
   - la partie publique dont le contenu est visible à l'extérieur du package
   - une partie privée visible depuis le package dans la partie privée et dans le corps
2. Le corps qui se trouve dans un fichier avec extension `.adb`. C'est dans ce fichier que les procédures et fonctions déclarées dans le fichier de spécification sont définies.

Le fichier de spécification ne peut pas contenir de corps (`body`). Par contre un package peut contenir un autre package. En ce qui concerne le package d'exemple ci-dessus, le type `Point` ne peut être manipulé directement. En effet, il faut utiliser les fonctions ou procédures définies dans la partie publique du package. Enfin, le corps du package peut contenir des exceptions qui servent à gérer les différents cas d'erreurs.

Les types dérivés et accès

Un type dérivé est basé sur un type quelconque et lui est presque identique. La dérivation de type inclut la copie des opérations primitives du type d'origine (héritage), et par conséquent on

[2] Gimp Toolkit
[3] GNU Image Manipulation Program

ne peut mélanger des éléments de deux types. Le type d'origine, appelé parent et le type dérivé appartiennent à la même catégorie de type : si le type parent est un article, le type dérivé est aussi un article.

Les types d'accès sont l'équivalent des pointeurs en C. A la base, les pointeurs contiennent l'adresse mémoire d'une donnée, que ce soit un entier ou le premier élément d'un tableau. Il en existe deux sortes :

- ceux permettant l'accès à d'autres éléments (ou données)
- ceux donnant l'accès à des sous-programmes

Ada peut être utilisé comme un langage orienté objet. En effet, il peut considérer l'héritage, l'encapsulation et le polymorphisme.

Le compilateur Gnat

Le compilateur Ada le plus utilisé est Gnat. Celui-ci se base sur le compilateur Gcc pour produire un code natif exécutable. Une table des symboles (les fichiers avec extension .ali) est créée pour chaque package afin d'accélérer la recherche des symboles dans les fichiers et donc détecter plus vite les erreurs de compilation. Gnat est livré avec une suite d'exécutables, dont le plus intéressant est `gnatmake`, sachant qu'à l'instar de javac (le compilateur Java) il compile un fichier cible et toutes ses dépendances. Il n'est donc pas forcément obligatoire de s'affranchir de l'écriture des Makefiles.

2.2.2 Gtk+

Gtk est un toolkit graphique conçu pour GNU/Linux qui a été développé pour la conception de Gimp en tant qu'alternative à Motif. Aujourd'hui Gtk est utilisé bien au-delà de Gimp et a suscité beaucoup d'engouement de la part des développeurs pour sa simplicité, sa modularité et ses bindings. Publié sous licence GPL[4], il est plébiscité dans de nombreux logiciels libres (l'environnement Gnome par exemple). Gtk+ est un ensemble de 3 librairies : Glib, Gdk et Gtk qui correspondent à trois couches d'utilité bien distinctes.

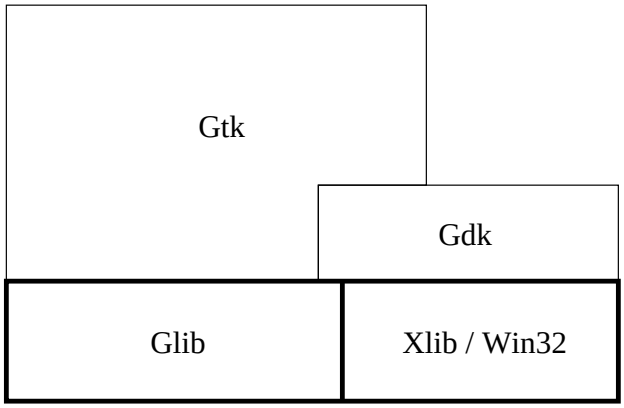


FIG. 2.5 – Organisation hiérarchique de Gtk+

[4]GNU Public Licence

Le rôle de chacune de ces librairies va être détaillé par la suite.

Glib

Glib est la couche la plus basse. Elle a les caractéristiques suivantes :
- Elle définit des structures de données qui sont souvent utilisées (listes chaînées, tables de hachage, analyseur lexical,...) et des fonctions permettant leur manipulation.
- Elle augmente la portabilité de Gtk+ et de ses applications en fournissant au développeur un accès transparent à certaines données, selon le système d'exploitation utilisé (par exemple `g_get_home_dir()` retourne le répertoire Home de l'utilisateur sous Linux et fournit un équivalent sous Windows.
- Gmodule est une composante de la Glib qui permet de charger dynamiquement des greffons ("plugins").
- Elle facilite le Débuggage avec une gestion des exceptions sous forme de macro-commandes.

Gdk

Gdk[5] est la couche intermédiaire de Gtk+. Son rôle est d'encapsuler les primitives de dessin du système (Xlib ou Win32) dans le cadre d'une application n'utilisant pas Gtk, soit pour permettre à Gtk de dessiner ses composants. Gdk affranchit le développeur de l'API[6] Xlib et optimise ses accès en libérant automatiquement les ressources allouées par le serveur X (pixmaps par exemple).

[5] Gimp Drawing Kit
[6] Application Programming Interface

Gtk

C'est la composante que beaucoup de monde tend à assimiler à Gtk+, sachant que la seule différence dans leur nom est le '+'. Gtk est la partie visible de l'iceberg car c'est le toolkit lui-même. Il possède bon nombre de composants graphiques (les widgets) qui suffisent à une application standard. Il existe plusieurs librairies aui en fournissent d'autres et il est de même très facile de concevoir ses propres widgets. Gtk est essentiellement une API orientée objet. Bien qu'elle soit entièrement écrite en C, elle est implémentée en utilisant la notion de classes et de fonctions de rappel (pointeurs de fonctions). Par exemple, la classe du widget `GtkLabel` hérite de la classe `GtkContainer` qui hérite elle-même de `GtkWidget` qui hérite aussi de `GtkObject`. Dès lors, on peut connecter des signaux à un label (puisque c'est un GtkObject), l'afficher (c'est un GtkWidget) et changer son contenu (c'est un conteneur), tout en pouvant changer ses propres caractéristiques.

Théorie des signaux

Gtk est dirigé par les événements, ce qui signifie qu'il restera inactif dans sa boucle principale (`gtk_main`) jusqu'à ce qu'un événement survienne et que le contrôle soit passé à la fonction de rappel appropriée.

Ce passage de contrôle est réalisé en utilisant le concept de signal. Lorsqu'un événement survient, comme l'appui sur un bouton, le signal approprié est émis par le widget qui á été pressé. C'est de cette façon que Gtk réalise la plupart de son travail. Pour qu'un bouton réalise une action, on configure un gestionnaire de signal pour capturer ces signaux et appeler la fonction adéquate. Ceci est fait en utilisant une fonction comme :

```
gint gtk_signal_connect ( GtkObject *object,
                          gchar* nom,
                          GtkSignalFunc fonction,
                          gpointer func_donnees);
```

Le premier paramètre est le widget qui émettra le signal, le deuxième est le nom du signal que l'on souhaite intercepter, le troisième est la fonction que l'on veut appeler quand le signal est capturé, et le dernier représente les données que l'on souhaite passer à cette fonction. La fonction spécifiée en troisième paramètre s'appelle une fonction de rappel (callback) et doit être de la forme :

```
1 void fonction (GtkWidget* widget,
2                gpointer donnee);
```

Le premier paramètre de cette fonction est un pointeur vers un widget et le second est un pointeur vers les données passées par le dernier paramètre de la fonction `gtk_signal_connect` décrite plus haut. Il est aussi possible de passer un objet Gtk en paramètre d'une fonction de rappel au lieu d'une variable de type abstrait. Dans ce cas, on utilise la fonction `gtk_signal_connect_object` avec une fonction de rappel de la forme :

```
1 void fonction( GtkObject* objet);
```

La valeur de retour des fonctions de connexion de signaux est de type `gint`. Il s'agit d'un marqueur qui identifie la fonction de rappel. Il est possible d'utiliser autant de fonctions de rappel que l'on veut, par signal et par objet, et chacune sera exécutée à son tour, dans l'ordre dans lequel elle à été attachée. Ce marqueur permet de déconnecter un signal de sa fonction de rappel :

```
1 void gtk_signal_disconnect (GtkObject* objet,
2                             gint id_rappel);
```

Le placement des widgets

Dans n'importe quel toolkit graphique, les widgets sont disposés hiérarchiquement dans des conteneurs (meta-widgets invisibles). Dans plusieurs d'entre eux, en revanche, les widgets sont positionnés de façon fixe dans leurs conteneurs. gtk offre cette possibilité; néanmoins, il est très rare que cette option soit choisie. Le placement automatique est en effet préféré par le biais de conteneurs de type boite ou table. La taille et/ou le placement des widgets est remis à jour si la taille de la fenêtre les contenant change. Évidemment, il existe aussi des fenêtres qui sont aussi des conteneurs ne pouvant stocker qu'un widget (un autre conteneur par exemple).

Les différents bindings

Une des caractéristiques qui fait la force de Gtk est d'avoir une API orientée objet, très homogène, mais écrite en C. Il est donc possible de l'utiliser depuis plusieurs langages, comme le C++, le Perl, le Python, l'Ada95 (cf partie sur GtkAda),...Ces "bindings" ne sont pas très durs à mettre en place car il ne constitue qu'une encapsulation des fonctions de Gtk, même si certains ajoutent/suppriment des fonctions ou fournissent de base pour des widgets supplémentaires. C'est notamment le cas pour GtkAda.

2.2.3 GtkAda : le binding Ada de Gtk+

GtkAda est l'adaptation en Ada de l'API de Gtk+. Avec ce toolkit, la création d'interfaces graphiques en Ada95 et multi-platformes (dont Unix et Win32) est aisée.

Bien que GtkAda soit basé sur son homologue en C, il utilise les possibilités de Ada telles que :
- les types taggés
- les packages génériques
- l'accès aux sous-programmes
- les exceptions

Pour des raisons d'efficacité, il n'utilise pas les types contrôlés, mais s'occupe efficacement de la gestion de la mémoire. Son fonctionnement étant similaire à celui de son homologue en C, seules les principales différences avec ce dernier seront détaillées.

Conventions de nommage

Le binding GtkAda est organisé en packages. En C, pour changer le texte d'un label, on appelle directement `gtk_label_set_text`. En Ada, il faut appeler la fonction `Set_text` du package `Gtk.Gtk_Label`

Les fonctions de rappel

Ces fonctions ne sont pas des callbacks comme en Gtk/C mais des *handlers*. Il faut de plus utiliser des *Marshallers* qui sont des intermédiaires entre le signal et sa fonction de rappel. Malgré le fait que certains marshallers soient définis, il faut quand même en définir pour chaque widget que l'on utilise. Ceux-ci font partie intégrante des handlers. Voici un exemple :

```
1  with Glib; use Glib;
2  with Gtk;
3  with Gtk.handlers;
4  with Gtk.Main;
5  with Gtk.Widget; use Gtk.Widget;
6  with Gtk.Window; use Gtk.window;
7
8  function Delete_Event_Handler (Window : gtk_Window) return Gint is
9  begin
10     -- Code execute a la destruction de la fenetre
11 end Delete_Event_Handler;
12
13 procedure Test is
14   package Window_Cb is new Gtk.Handlers.Callback (Gtk_Window_Record);
15   fenetre : Gtk_Window;
16 begin
17  Gtk.Main.Init;
18
19  Gtk_New(fenetre);
20  window_Cb.Connect( Fenetre,''delete_event'',
21                     Window_Cb.To_Marshaller (Delete_Event_Handler));
22  show(fentre);
23
24  gtk.Main.Main;
25 end Test;
```

La gestion des signaux

Contrairement à Gtk/C, on ne peut pas passer des données quelconques (`gpointer donnees`) aux handlers. En GtkAda, *Connect* ne permet à la fonction de rappel que de recevoir en paramètre l'objet qui a émis le signal.

La solution est d'utiliser `Object_Connect` qui permet d'obtenir au niveau de la fonction de rappel toujours un seul paramètre, mais qui est un objet différent de celui qui a émis le signal. Cela ne règle pas tout le problème, mais la solution a été complétée en ajoutant les données nécessaires dans l'objet considéré.

Afin de pouvoir travailler efficacement en équipe, CVS a été utilisé. CVS est un outil de gestion de versions de fichiers. Il est beaucoup utilisé au cours du développement car il permet de sauvegarder efficacement le travail de chacun dans un dépôt centralisé. Ainsi, Cheddar évolue continuellement au cours du développement. Les avantages et inconvénients de cet outils sont argumentés dans la section suivante.

2.2.4 CVS

Desciption de CVS

CVS (Concurent Versions System) permet la gestion des sources d'un projet, un outil de suivi de version. Il permet de conserver trace de l'historique des modifications d'un fichier, ou d'un ensemble de fichiers, et de revenir simplement à n'importe quel état antérieur.

Pour garder cet historique, il serait bien sur possible de conserver chaque version de chaque fichier que vous créez ou manipulez. Cela prendrait évidemment une place disque considérable. L'intérêt d'utiliser un outil comme CVS est d'abord qu'il stocke seulement les différences entre deux versions successives, réalisant ainsi un gain de place appréciable. Ensuite, CVS enregistre automatiquement la date, l'auteur et un éventuel commentaire explicatif pour chaque modification.

Un autre avantage de CVS est qu'il facilite le travail en groupe. En effet, travailler à plusieurs dans un même répertoire demande une organisation sans faille et une attention de tous les instants afin de ne pas écraser les modifications faites par un autre membre du groupe. CVS permet à chacun de travailler dans son propre répertoire, même éventuellement sur sa propre machine et prend en charge l'intégration du travail des différents membres du groupe.

Notons également que CVS est un logiciel libre, diffusé sous licence GPL. Il fonctionne sur une grande variété de machines et de systèmes d'exploitation : Linux, Solaris, BSD, MacOS, MS-Windows, etc.

Principe de CVS

CVS stocke tout l'historique des évolutions d'un fichier sous forme compacte dans un fichier historique. Les fichiers historiques sont conservés dans des repositories.

Chaque fois qu'un utilisateur souhaite travailler sur un fichier ou un ensemble de fichier, il demande à CVS de produire, à partir du contenu du repository adéquat, une copie privée des fichiers souhaités dans la version désirée. Cette opération est appelée checkout dans la terminologie CVS. C'est sur cette copie personnelle, appelée copie de travail que l'utilisateur effectue ses modifications, tests, etc.

Une fois satisfait de son travail, l'utilisateur demande à CVS de mettre à jour le repository en y incorporant les modifications qu'il a réalisées sur sa copie de travail. Cette opération est appelée commit par CVS. L'utilisateur peut ensuite continuer à travailler avec sa copie personnelle et demander à CVS d'incorporer ses modifications au repository chaque fois qu'il le souhaite.

Une fois son travail terminé, si un utilisateur ne souhaite pas conserver sa copie de travail, il peut l'effacer et signaler à CVS qu'il ne possède plus de copie personnelle, grâce à un release. Il pourra de toute façon à tout moment, recréer une copie de travail à partir du repository.

Dans le cas le plus général, plusieurs utilisateurs peuvent posséder une copie de travail des même fichiers, et faire chacun des modifications qui leur sont propres. Pour le premier qui demandera à CVS d'intégrer ses modifications dans le repository, tout se passera comme s'il était seul. Mais lorsque le deuxième utilisateur incorporera ses modifications dans le repository, CVS se rendra compte que ces modifications ont été portées sur une version plus ancienne que la version disponible dans le repository et combinera ces modifications à celles effectuées par le premier utilisateur. Il en profitera également pour mettre à jour la copie de travail du deuxième utilisateur.

Il est possible cependant que CVS ne sache pas comment combiner deux ensembles de modifications successives, par exemple si le deuxième utilisateur a modifié une ligne supprimée par le premier. Dans ce cas, il indique au deuxième utilisateur les fichiers en conflit et lui laisse le soin de résoudre manuellement le problème.

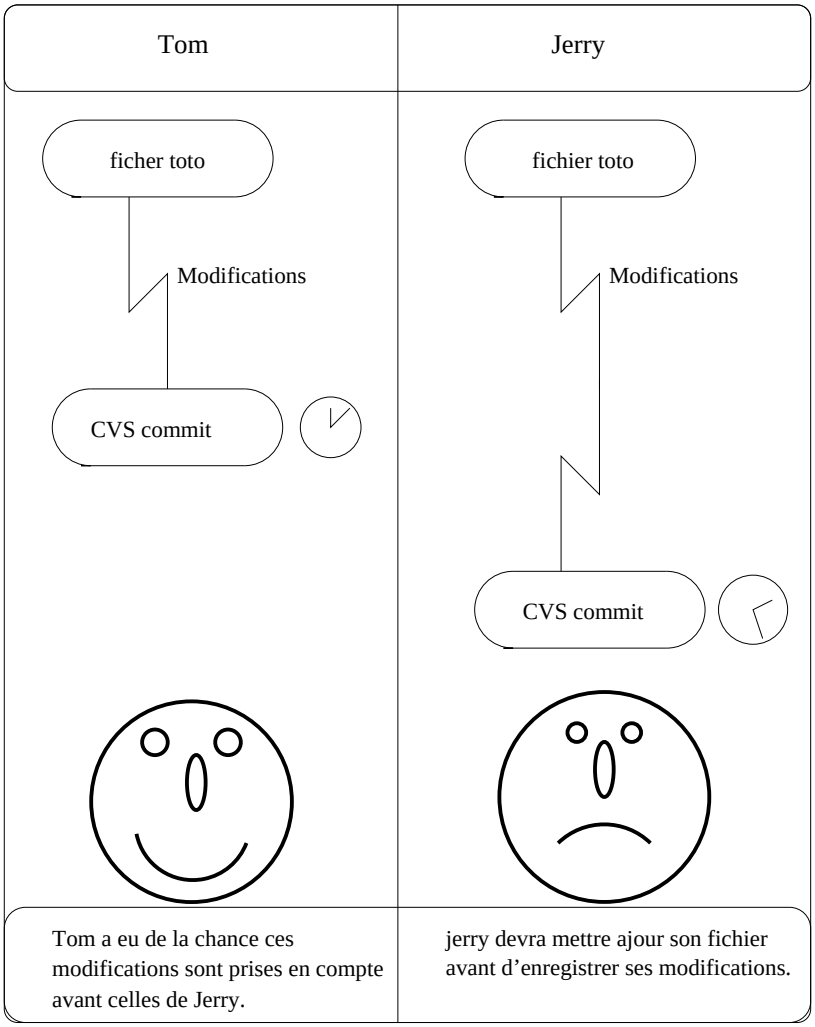


FIG. 2.6 – illustration du manque de dialogue

Il est à noter que le premier utilisateur n'est pas informé automatiquement des modifications apportées par le deuxième sur les fichiers sur lequel il travaille. S'il ne fait rien de particulier, il n'en aura connaissance que lors de son prochain commit. Tout utilisateur possédant une copie de travail peut cependant, à tout moment, demander à CVS de la mettre à jour par rapport à la dernière version disponible dans le repository, autrement dit d'incorporer dans sa copie de travail toutes les modifications apportées par tous les utilisateurs travaillant sur les mêmes fichiers que lui. Cette opération s'appelle un update. Bien entendu, le dialogue entre programmeurs peut éviter ces petit désagrément.

Commande de CVS

CVS offre un certain nombre de commandes aux programmeur. Les plus utilisées peuvent être au nombre de cinq :

- **CVS checkout**
  consiste a créer un copie local d'un ensemble de fichier ins crit dans le repository.

- **CVS update**
  permet la mise a jour des fichiers locaux par rapport au repository.
- **CVS commit**
  synchronise la version locale du projet ou d'un de ces modules avec le repository.
- **CVS add**
  permet l'ajout de l'entrée d'un fichier dans le repository, cette action doit être suivie d'un CVS commit pour que la version local du fichier soit prise en compte.
- **CVS remove**
  ne supprime pas les versions anterieures du fichier concerné mais lors des update ou commit suivant ce fichier ne sera pas pris en compte.

2.3 Avancement du projet

2.3.1 Le projet début juin

Au début du stage voici les fonctionnalités implantées dans cheddar :

- Ajouter / Supprimer / Mettre a jour un processeurs.
- Ajouter / Supprimer / Mettre a jour une tache.
- Tracer un ordonnancement
- Contrôler la faisabilité d'un ordonnancement
- Sauvegarder un projet (Seulement tache et processeur)
- Charger un projet.
- Nettoyer l'espace de travail.
- Algorithmes d'ordonnancements implantés : RM, DM, EDF, LLF, HPF

Il nous a été demandé d'améliorer les fonctionnalité existantes et d'en ajouter.

2.3.2 Amélioration

Cheddar est compilable sous Windows, Linux et Solaris. Le seul problème au cours de la compilation est que certains fichiers (ceux de l'internationalisation du logiciel) se compilent très lentement. Ceci est dû au nombre très important de variables temporaires :

```
Lb_Root_Title(Francais) := To_Unbounded_String(
   "Cheddar : un simulateur d'ordonnancement temps réel");
Lb_File(Francais) := To_Unbounded_String("Fichier");
Lb_New(Francais) := To_Unbounded_String("Nouveau");
Lb_Open(Francais) := To_Unbounded_String("Ouvrir");
```

Il y a trop de `To_Unbounded_String()`, ce qui provoque un warning du compilateur Ada et un contretemps énorme en phase de déboggage. A ce problème, deux solutions ont été proposées :

- l'utilisation de **gettext** a été envisagée. Gettext permet de remplacer des chaînes de caractères d'une certaine langue par d'autres de langues différentes lors de l'exécution du programme, à la volée. Une macro ('-') est utilisée afin de préfixer toutes les chaînes de caractères du programme. Ensuite le programme *xgettext* permet d'extraire ces chaînes et de constituer un fichier de référence avec un ensemble de couples (msgid, msgstr). `msgid` correspond à la chaîne à traduire et `msgstr` sa traduction dans une certaine langue. Il suffit de copier ce fichier, de le renommer *fr.po* pour le français par exemple, et de remplir les lignes `msgstr`. Hélas, cette solution a des lacunes au niveau de la portabilité et n'a donc pas été adoptée.
- l'utilisation d'une énumération à double dimension (pour français et anglais). Cette solution a l'avantage d'être plus pratique d'utilisation mais ne fait que contourner le problème. En effet le nombre de variables locales reste le même. Toutefois, elle a été mise en oeuvre mais pas commitée sur le CVS.

2.3.3 Planning

Ci-dessous le planning prévisionnel du stage, durant ces deux mois il à été suivi et à permis de respecter les délais.

Groupes / Semaines	Mise en place des précédences	Ressources partagées
Semaine 1	Apprentissage de l'Ada et GtkAda	Apprentissage de l'Ada et GtkAda
Semaine 2	Canvas et toolbar	Théorie sur PCP / PIP RM, DM Création menu / fenêtre
Semaine 3	Canvas et toolbar Finalisation de la toolbar Fenêtre de vue des messages	Analyse Scheduler Cheddar =¿ greffe Création fenêtre d'ajout / suppression de ressources Programmation de l'objet ressource
Semaine 4	Fenêtre de création de messages Mise en place des dépendances Fenêtre de création de buffers	Programmation PCP Création fenêtre de mise à jour
Semaine 5	Débuggage fenêtre de création de buffers Fenêtre de vue des buffers Gestion des dépendances	Programmation PIP Débuggage fenêtre de mise à jour Fenêtre de vue
Semaine 6	Mise en place d'une nouvelle mise à jour des taches Semi-dépendences Mise en place de la mise à jour des buffers	Modification fenêtre de mise à jour Mise en place de blocking Time, compute et inject
Semaine 7	Débuggage IHM tache Débuggage dépendances Débuggage de l'IHM buffer Débuggage IHM message Nettoyage du code Rapport	Sauvegarde ressource Nettoyage du code Débuggage PCP Débuggage PIP Débuggage IHM Rapport
Semaine 8	Rapport	Rapport

Chapitre 3

La tache réalisée

Un certain nombre de fonctionnalités et d'améliorations ont été apportées à Cheddar dont les principales sont :

- l'ajout du concept de ressources
- l'ajout d'algorithmes d'ordonnancement
- l'ajout d'une vue 'graphe' à Cheddar
- l'interfacage d'objets existant dans Cheddar
- le ré-interfaçage de certains objets de Cheddar

Ces modifications et ajouts sont expliqués en détails dans les sections suivantes.

3.1 Les ressources partagées

Nous allons introduire ici le concept de ressource, nous parlerons ensuite des algorithmes de partage utilises pour la gestion de ces ressources, et enfin le L'interface homme machine qui permettent a l'utilisateur de les utiliser.

3.1.1 Le concept de ressource

Définition

Une ressource est une structure logicielle pouvant être utilisée par une tache pour avancer dans son exécution. Par exemple un ensemble de variable la mémoire partagée, un fichier etc. Une ressource est privée si elle est dédiée a une tache particulière et partagée si elle est utilisée par plusieurs tache. Dans ce dernier cas des sémaphores assurent l'accès exclusif.

Le partage de ressources dan un système temps réel peut être a l'origine du phénomène d'inversion de priorité. Pour éviter ce genre de problème il est possible d'utiliser des protocole de partage de ressource dont le fonctionnement sera étudie plus tard.

Le partage de ressource permet a des taches différentes de pouvoir exploiter les mêmes informations en même temps et de manière transparente pour l'utilisateur. Il est donc possible pour deux taches de lire le même fichier ou la même zone mémoire par exemple.

Implémentation

Dans Cheddar une ressource peut être gérée par deux protocoles différents ou par aucun. Un ressource possède dons un protocole attache, celui ci peut être PIP,PCP ou No_Protocole (Pas de protocole). Un type Ressource_type a donc été définis, celui ci permet de connaître le type de protocole utiliser et de savoir quelle méthode est adaptée au traitement de cette ressource.

```
1   type Resources_Type is (
2              No_Protocol,
3              Priority_Ceiling_Protocol,
4              Priority_Inheritance_Protocol);
```

Une ressource est définie par son nom, son état (compteur d'accès) un protocole utilise pour la gestion de cette ressource une liste de tache qui lui accède, et un processeur auquel elle est rattachée. Un type Generic_ressource a été déclare pour symboliser cet objet.

```
1  type Generic_Resource is abstract new Ada.Finalization.Controlled with
2      record
3           Name       : Unbounded_String := To_Unbounded_String(");
4           State      : Natural          := 0;
5           Protocol   : Resources_Type;
6           Task_Tab   : Task_List;
7           Cpu        : Unbounded_String := To_Unbounded_String(");
8      end record;
```

Il est indispensable pour une tache de connaître les taches qui y accède. Il y avait deux possibilités pour cela. La première était de stocker dans chaque tache la liste des ressource quelle utilise ainsi que le début et la fin d'accès en unité de temps. La seconde était de stocker dans la ressource elle-même le nom de la tache qui y accède ainsi que le début de son accès et la fin.

C'est la seconde solution qui a été retenue car pour l'étape suivant (Implémentation des protocole PIP et PCP) les algorithmes sont simplifies. Un type Affected_Task_List a donc été défini. Ce type permet de connaître le nom de la tache qui accède a la ressource, le temps de début de son accès et le temps de fin de son accès (en unités de temps). Ce type sera stock dans un tableau de Max_Tasks éléments. Par défaut Max_Tasks est égal a cinquante ce chiffre après mure réflexion nous a paru correcte, ceci voulant dire qu'une ressource peut avoir cinquante accès différents.

```
1      type Affected_Task_List is
2      record
3        Task_Name       : Unbounded_String := To_Unbounded_String(");
4        Task_Begin      : Natural          := 0;
5        Task_End        : Natural          := 0;
6      end record;
7
8      type Task_List is Array (Natural Range 1..Max_Tasks) of Affected_Task_List;
```

L'objet générique ressource a été décline en plusieurs sous objet PCP_Ressource, PIP_Ressource et NP_Ressource.

```
1   type PCP_Resource is new Generic_Resource with private;
2
3   type NP_Resource is new Generic_Resource with private;
4
5   type PIP_Resource is new Generic_Resource with private;
```

Chacun de ses objets disposes de méthodes d'initialisation et de déclaration de pointeurs. Ces pointeurs permettent d'utiliser les taches lors de recherches dans le set qui les contient.

En effet, tous les objet ressources sont stockes dans une liste qui n'est pas encore dynamique mais devrai le devenir. Des méthodes d'actes aux éléments de cette liste on été crée dans le package set.

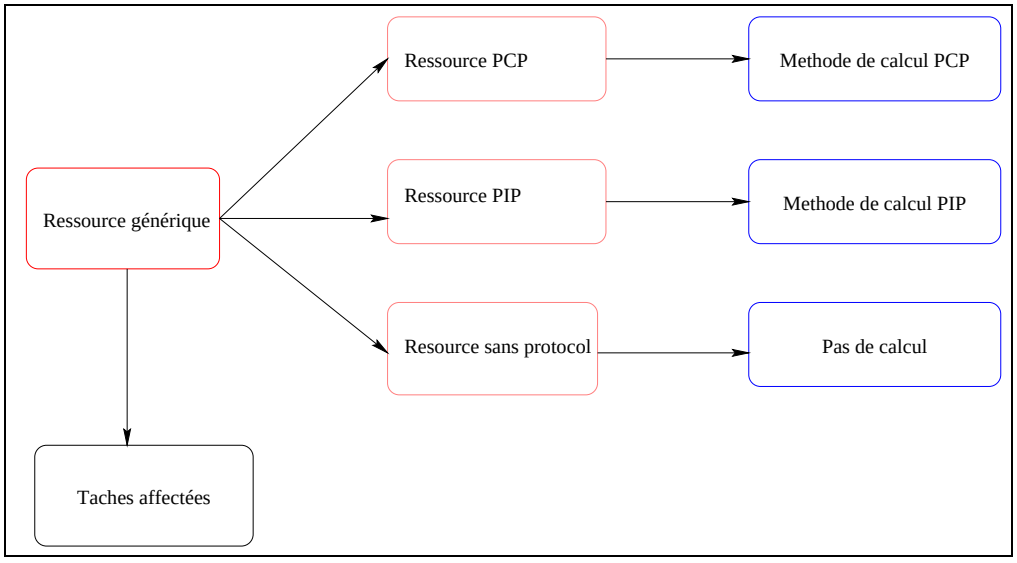


FIG. 3.1 – L'objet ressource

3.1.2 Les algorithmes de partage

Les problèmes d'inversion de priorité

Il existe des algorithmes classiques d'ordonnancement optimaux pour les systèmes de taches indépendantes, mais comme tout système multitâche, un système temps réel doit permettre aux taches d'accéder a des ressources, qu'elles ont a partager. Nous présenterons ici un cas connu sous le nom d'inversion de priorité qui montre que les résultats d'optimalité de certains algorithmes préemptifs utilisant des priorité ne s'appliquent plus dans le cas de partage de ressources.

Les algorithmes a priorités statique

Considérons un système de trois taches t_1,t_2,t_3 ayant comme priorité P_1<P_2 <P_3 (P_3 étant la tache la plus prioritaire, et P_1 la moins prioritaire).Supposons que t_2 et t_3 aient fini leur exécution et attendent la fin de leur période respective pour être réactivées. t_1 s'exécute en utilisant une ressource R. t_2 est réactivées et interrompt $t1$ qui n'a pas libère R.t_3 est réactivées, et bien que plus prioritaire que les deux autres taches, elle doit attendre la terminaison de t_2 qui bloque t_1, puis que t_1 libre R.

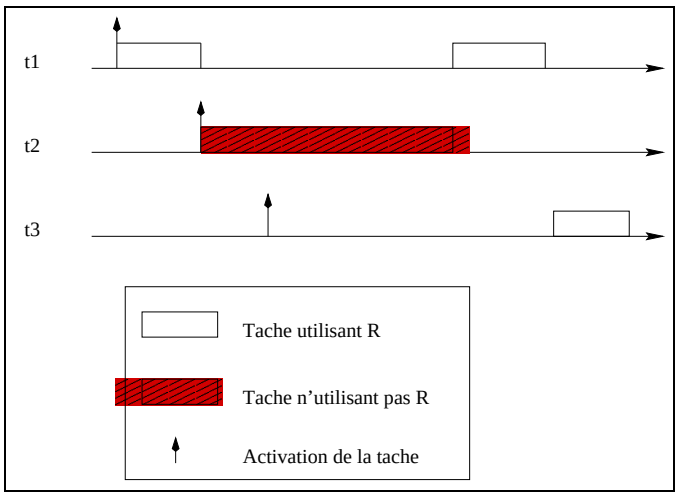


FIG. 3.2 – Inversion de priorité

Il n'existe pas a ce jour d'algorithmes polynômial permettant d'ordonnancer les systèmes de taches partageant les ressources.

Les algorithmes a priorités dynamiques

Le même exemple sera utilisé, mais cette fois les priorités ne seront pas fixe au départ. Considérons l'algorithme ED brièvement décrit dans la partie 2, et supposons les échéances des taches, notées R_1,R_2 et R_3, telles que $R_1 > R_2 > R_3$. On se retrouve dans le même cas se figure que pour les algorithmes a priorités statique. Il existe une variante de ED qui utilise un protocole a priorité héritées qui gère les inversion de priorité de la manière suivante : quand une tache est bloquante, on lui donne la priorité de la tache qu'elle bloque qui possède la plus haute priorité, et ce jusqu'à la libération de la ressource bloquante. Cette méthode évite les inversions de priorités mais sont fonctionnement n'est pas optimal.

PIP - Priority Inheritance Protocol

Introduit par Sha, Rajkumar et Lehoczky en 1990, le protocole PIP empêche les taches de priorité moyennes de préempter la tache responsable du blocage d'une tache de plus haute priorité et de prolonger ainsi son temps de blocage.

Principe

L'idée de base de PIP est d'affecter temporairement la tache responsable du blocage avec la priorité de la tache la plus prioritaire. Chaque tache J_i possède une priorité fixe (déterminé par RM) ainsi qu'une priorité actuelle $P(t)_i$. Lorsqu'une tache J_i demande une ressource R_k, on a l'une des situations suivantes :

- Si R_k est libre J_i prend la ressource et entre en section critique
- Si R_k n'est pas libre ; J_i est bloquée. De plus, la tache qui bloque J_i hérite de sa priorité actuelle. Cette tache continue à s'exécuter avec sa nouvelle priorité jusqu'à ce qu'elle libère la ressource. Elle reprend alors sa priorité initiale (c'est a dire celle qu'elle avait avant de prendre la ressource).

L'héritage de la ressource étant transitif, il se peut qu'une tâche hérite plusieurs fois si elle bloque plusieurs taches.

Considérons une ressource R_0 partagée entre des taches J_0, J_1 et J_2 telles que $P_0 > P_1 > P_2$.

Sans PIP ceci donnerai :

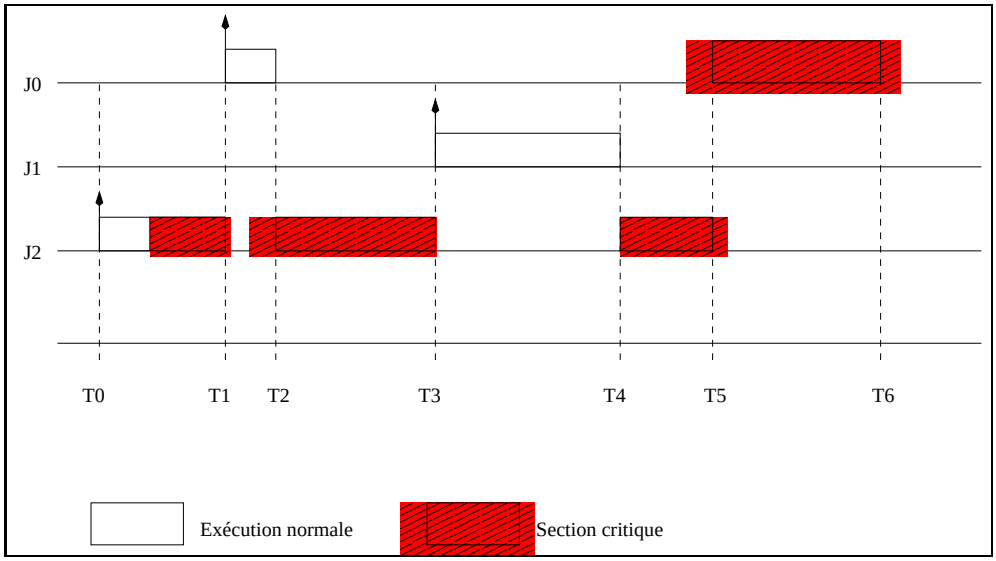


FIG. 3.3 – Ordonnancement sans PIP

- T_0 : J_2 est activée, peut prendre R_0 et entre en section critique.
- T_1 : J_0 préempte J_2 et commence son exécution.
- T_2 : J_0 ne peut prendre R_0 car la ressource n'est pas libre. J_2 reprend son exécution.
- T_3 : J_1 est activée et préempte J_2.
- T_4 : J_2 ne peut prendre R_0 car la ressource n'est pas libre. J_2 reprend son exécution.
- T_5 : J_2 libère la ressource R_0, J_0 préempte J_2 et entre en section critique en prenant la ressource. La durée du blocage du a l'inversion de priorité est égale a $T_5 - T_2$: elle peut être beaucoup plus importante que la durée de la section critique de J_2 car cette tache peut-être préemptée par d'autres taches de priorité supérieure.
- T_6 : J_0 libéré R_0.

Avec PIP on a :

- T_0 : J_2 est activée, peut prendre R_0 et entre en section critique.
- T_1 : J_0 préempte J_2 et commence son exécution.
- T_2 : J_0 ne peut prendre R_0 car la ressource n'est pas libre. J_2 reprend son exécution avec la priorité de J_0.
- T_3 : J_1 est activée mais sa priorité est désormais inférieure a celle de J_2 qui continue son exécution. On voit ici que PIP introduit un nouveau type de blocage, le blocage d'héritage de priorité.
- T_4 : J_2 Libère la ressource , reprend sa priorité initiale et est préemptée par J_0. J_0 peut prendre la ressource et rentre en section critique.
- T_5 : J_0 termine son exécution en libérant la ressource R_0, J_1 peut commencer son exécution.

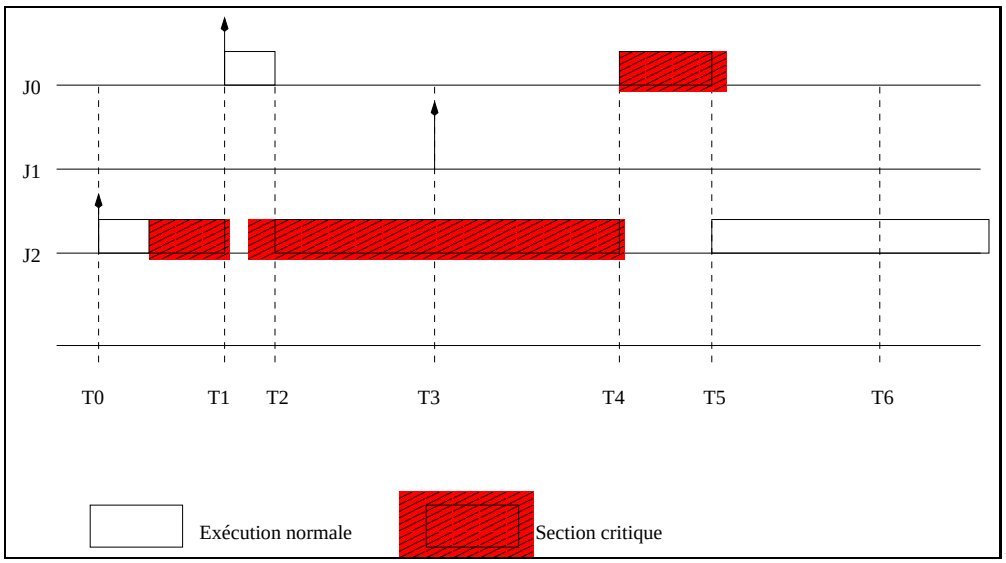


FIG. 3.4 – Ordonnancement avec PIP

PIP n'empêche pas les situations d'inter-blocage, qui peuvent survenir lorsqu'il y a plusieurs ressources partagées :

Prenons une tache J_0 qui attend une ressource R_0 puis une autre R_1 avant de libérer R_0, une autre tache J_1 effectue l'allocation de ressource inverse , avec $P_0 > P_1$.

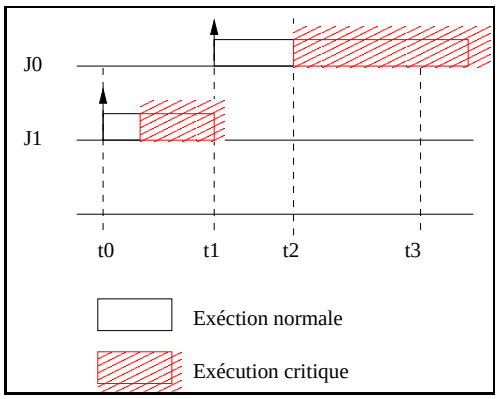


FIG. 3.5 – Inter-blocage avec PIP

- T_0 : J_1 est activée, peut prendre R_0 et entre en section critique
- T_1 : J_0 est activée et préempte J_1
- T_2 : J_0 peut prendre R_1 car cette ressource est libre
- T_3 : J_0 ne peut pas prendre R_0 car cette ressource est utilisée par J1. De même J_1, ne peut pas prendre R_1 et on a une situation d'inter-blocage

De plus PIP peut entraîner le phénomène de chaîne de blocage, qui survient lorsque la tache la plus prioritaire est bloquée une fois par toutes les autres taches. Considérons trois taches J_0,J_1,J_2 avec $P_0 > P_1 > P_2$, ainsi que deux ressources R_0 et R_1. J_0 utilise R_0 puis R_0 puis R_1,J_1 utilise R_1 et J_2 utilise R_0.

Calcul des temps de blocages

Le calcul des temps de blocages avec PIP se fait grâce aux d'équations suivantes :
B_i étant le temps de blocage de la tache i. P_i étant la priorité d'une tache i. D_j,k étant la durée de la section critique d'une tache j, pour la ressource k.

$$\begin{cases} \mathcal{B}_i^1 = \sum_{j=i+1}^{n} \max_k \{D_{j,k} : \prod(Rk) \geq P_i\} \\ \mathcal{B}_i^s = \sum_{k=1}^{m} \max_{j<i} \{D_{j,k} : \prod(Rk) \geq P_i\} \\ \mathcal{B}_i = \min\{B_i^1, B_i^s\} \end{cases}$$

Le calcul du temps de blocage se divise en trois étapes :
Prenons une tache T_i

- Étape 1 : L'ensemble des sections critiques pouvant bloquer T_i est recherché pour chaque tache T_n ayant une priorité inférieure a P_i (priorité de T_i). La plus grande des sections critique pour cette tache est retenue. La somme de ces valeurs est calculée. En parallèle le nombre de tache qui on au moins une section critique avec T_i sont comptabilisé.
- Étape 2 : Pour chaque ressource R_n, l'ensemble des sections critiques S_k qui utilisent cette ressource et peuvent bloquer T_i sont recherché. La durée maximale dans cet ensemble est retenue pour la ressource R_n. La somme de ces valeurs est calculée. En parallèle on compte le nombre de taches qui peuvent bloquer T_i.
- Ensuite le nombre de taches qui bloque T_i est multiplié avec la durée maximale de la section critique calcule dans la première étape. Par analogie le nombre de ressource bloquant T_i est multiplié avec la somme des section critiques pour les taches R_k. Le minimum de ces deux valeurs est gardée.

Voici un exemple avec la première et la deuxième étape décrites ci-dessus Prenons pour le premier exemple $P_2 < P_1 < P_3 < P_4$ (c'est a dire que la tache prioritaire est T_2).Les taches moins plus prioritaires que T_2 n'aurait pas été considérées dans l'algorithme. La tache utilisée pour l'algorithme est T_2 Pour le second exemple on aura $P_1 < P_2 < P_3$ (P_1 est la plus prioritaire). La tache considérée est ici T_1.

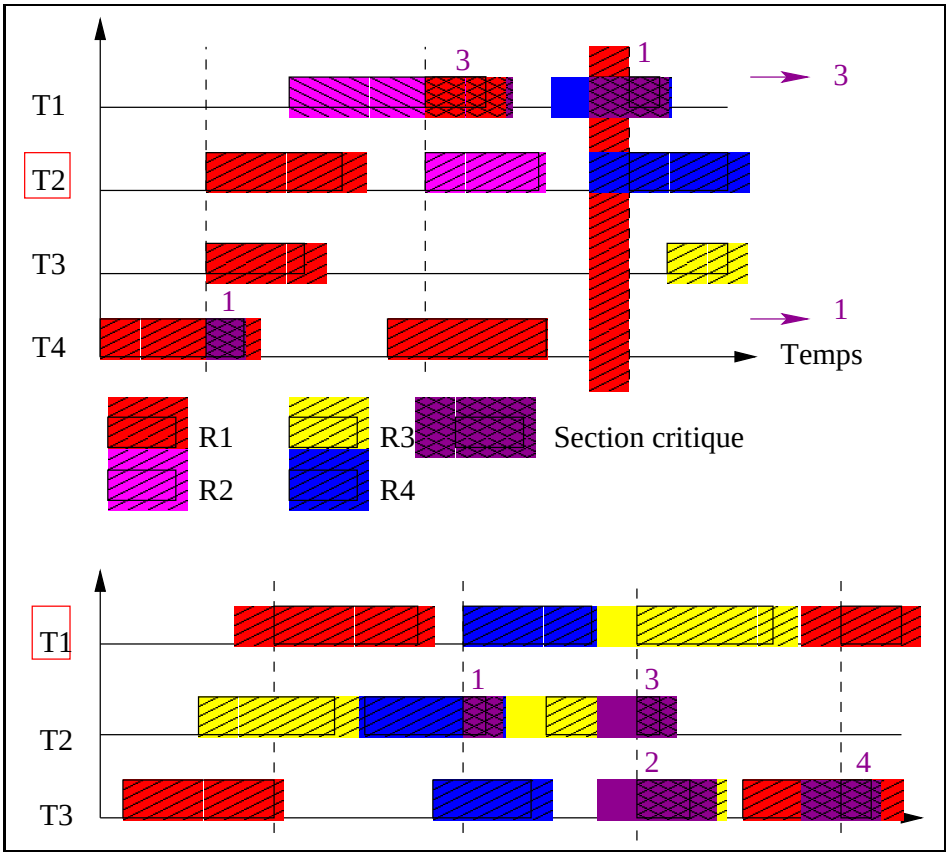


FIG. 3.6 – Temps de blocage avec PIP

Dans le premier exemple il y a trois sections critiques, une de 3 unités de temps avec T_1, une autre d'une unité de temps avec T_1, et une d'une unité de temps avec T_4. Pour la tache T_1 il y aura donc un temps de blocage de 3 ($\max(1, 3)$) et de 1 pour T_4. Dans ce cas il y a deux taches qui peuvent bloquer T_2. Le résultat obtenu pour la tache considérée est :

Pour les taches :

$$3 + 1 = 44 * 2 = 8$$

Pour les ressource :

$$3 + 4 = 77 * 3 = 21$$

Le résultat final :

$$\min(8, 21) = 8$$

Le temps de blocage de T_2 est de 8 unités de temps.

Les algorithmes ADA

Voici une brève descriptions des algorithmes développés.

```
1    PIP_Critical_Tasks(Task_Name,A_Task_Set,A_Resource_Set,Tasks_Critical,Tasks_N);
```

- Task_Name : Le nom de la tache pour laquelle l'algorithme PIP se déroule.
- A_Task_Set : L'ensemble des taches su système.
- A_Resource_Set : L'ensemble des ressources du système.
- Tasks_Critical : La somme des section critiques calculée.
- Tasks_N : Le nombre de taches pouvant bloquer la tache Task_Name.

Cette fonction retourne Task_critical et Task_N. Elle permet de connaître la sommes des section critiques pour une tache Task_Name, et le nombre de taches qui peuvent la bloquer. Elle correspond a l'étape une vue précédemment.

```
1    PIP_Critical_Resources(Task_Name,A_Task_Set,A_Resource_Set,Resources_Critical,Res
```

- Task_Name : Le nom de la tache pour laquelle l'algorithme PIP se déroule.
- A_Task_Set : L'ensemble des taches su système.
- A_Resource_Set : L'ensemble des ressources du système.
- Resources_Critical : La somme des section critiques calculée pour les ressources.
- Resources_N : Le nombre de ressources pouvant bloquer la tache Task_Name.

Cette fonction retourne Resources_Critical et Resources_N. Elle permet de connaître la somme des section critique des ressource pour une tache Task_Name et le nombre de ressource pouvant la bloquer. Elle correspond a l'étape deux vue précédemment.

Ces deux fonctions sont ensuite utilisées dans la fonctions suivantes :

```
1  function Blocking_Time ( Task_Name           : in Unbounded_String;
2                           A_Task_Set          : in Tasks_Set;
3                           A_Resource_Set      : in Resources_Set
4                         ) return Natural is
```

- Task_Name : Le nom de la tache pour laquelle les deux algorithmes précédent seront exécutes.
- A_Task_Set : L'ensemble des taches du système.
- A_Resource_Set : L'ensemble de ressources du système.
- retourne : Le temps de blocage de la tache Task_Name.

```
1  procedure Make_Table (Object_Name : in Unbounded_String;
2                        Max       : in Natural;
3                        N       : in out Natural;
4                        Table     : in out Temp_Table) is
```

- Object_Name : Nom de l'objet a stocker

- Max : Valeur correspondante a l'objet a stocker
- N : Nombre d'éléments dans le tableau ou sont stocké les objets
- Table : La table dans laquelle sont stocké les objets.

Cette procedure permet de stocker dans un tableau Table des objets de nom Object_Name et leur valeur associées Max. N est mis à jour à chaque ajout d'élément dans la table.

Si Object_Name n'existe pas lors de l'appel de cette procedure, il sera ajouté dans la table, avec sa valeur Max.

Si Object_Name existe la valeur Max est mise a jour si celle-ci est inférieure a la valeur Max passée en paramètre.

Ce système permet de stocker n'importe quel type d'objet ayant une valeur associée.

Pour permettre ceci une structure Temp_table a été définie. Celle-ci est un tableau de structure de Max_Tasks éléments.

PCP - Priority Ceiling Protocol

Principe

Définis par Sha, Rajkumar et Lehoczky en 1987, PCP est une amélioration de PIP permettant d'éviter les situations d'interblocage ainsi que les chaînes de blocage. Le support d'ordonnancement est RM.

L'idée de base de PCP est d'étendre PIP avec des règles pour prendre les sémaphores , ces règles interdisant une tache d'entrer en section critique s'il y a des sémaphores bloqués susceptibles de bloquer cette tache. Ce protocole suppose que chaque tache possède une priorité fixe et que les ressources utilisées sont connues avant le début de l'exécution. Ce protocole est une protocol a plafond de priorité.

Le Plafond de priorité d'une ressource Ri est la plus haute priorité des taches accédant a Ri, et on la note $\prod(R_i)'$. Le plafond de priorité courant su système, noté $\prod$ est égal au maximum des plafonds de priorité des ressources utilisés, ou Ω si aucune ressource n'est utilisé (Ω est une priorité plus base que n'importe qu'elle priorité)

Considérons trois taches J_0,J_1 et J_2 avec $P_0 > P_1 > P_2$ ainsi que trois ressources partagées R_0,R_1 et R_2 telles que :

- J0 utilise R0 puis R1
- J1 utilise R2
- J2 utilise R2 puis R1

On a donc $\prod(R_0) = P_0$, $\prod(R_1) = P_0$ et $\prod(R_2) = P_1$

- T_0 : J_2 est activé peut prendre R_2 car $P_2 > \prod' = \Omega$.
- T_1 : J_1 préempte J_2.
- T_2 : J_1 ne peut prendre R_2 car P_1 n'est pas $> \prod' = \prod(R2)$. J_2 hérite de la priorité de J_0 et reprend son exécution.
- T_3 : J_2 peut prendre R_1 car aucune ressource n'est utilisé.
- T_4 : J_0 préempte J_2
- T_5 : J_0 ne peut prendre R_1 car aucune ressource n'est utilisée.

- T_6 : J_2 libère R_1 et reprend sa priorité $P_2 = P_1$. J_2 est alors préempté par J_0 et ce dernier prend R_0 car on a $P_0 > \prod' = \prod(R_2)$.
- T_7 : J_0 termine son exécution J_2 reprend la sienne et prend la ressource R_2.
- T_8 : J_2 libère R_2 et reprend sa priorité initiale P_2 : J_1 préempte J_2 et prend R_2 car on a $P_1 > \prod' = \Omega$
- T_9 : J_1 termine son exécution, J_2 peut reprendre et terminer la sienne.

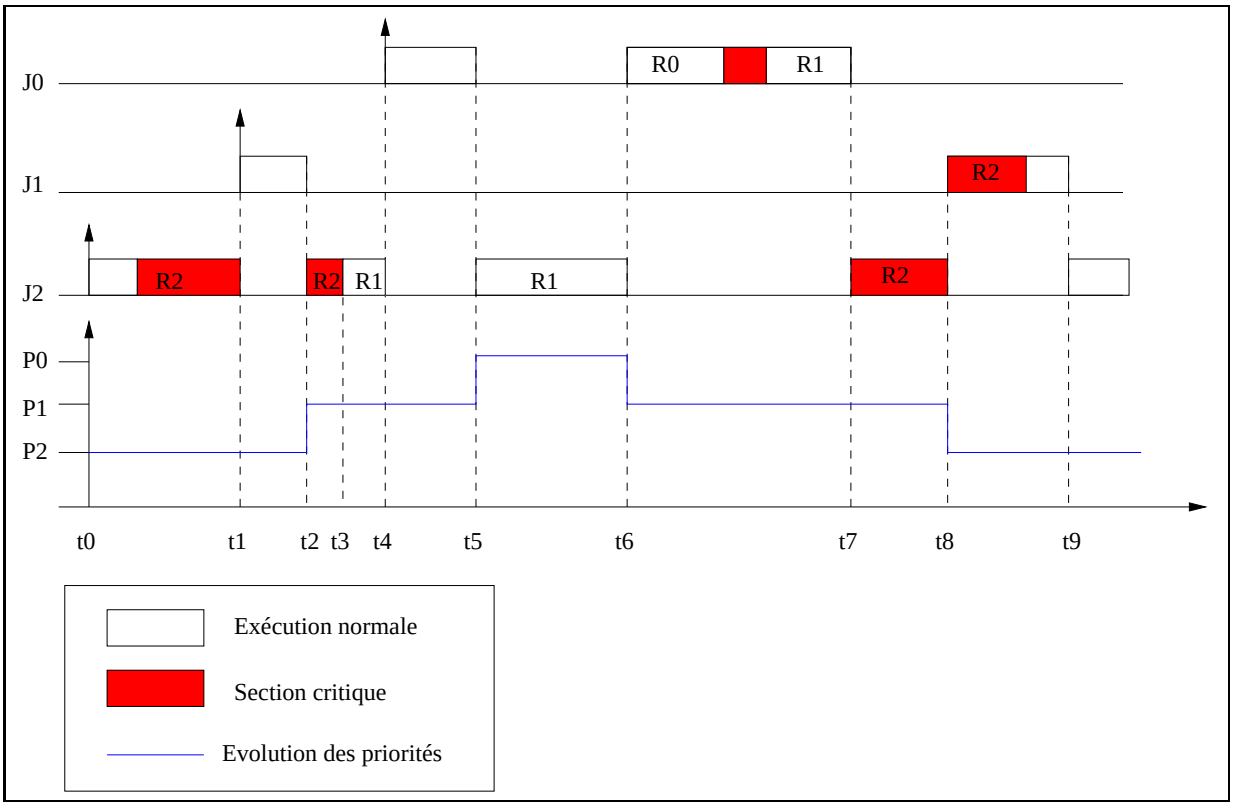


FIG. 3.7 – Application de PCP

Le temps de blocage avec PCP

Avec PCP, le temps de blocage est réduit à la durée de la plus longue section critique. La difficulté est d'identifier l'ensemble des sections critiques susceptibles de bloquer une tache. Pour une section critique exécutée par une tache J_i utilisant une ressource R_k, on montre qu'elle peut bloquer une tache J_i seulement si P_j $textless P_i$ et $\prod(R_k) \geq P_i$. Le temps de blocage est donc $B_i =$ durée de la plus grande section critique parmi celles des taches de priorité $\leq P_i$ et dont la ressource relative R_i est telle que $\prod(R_i) \geq P_i$.

Prenons $D_(j,k)$ la durée de la section critique de la tache J_j utilisant R_k on a :

$$\mathcal{B}_i = \max_{j,k}(D_(j,k) : P_j < P_i et \prod(R_k) \geq P_i)$$

nb : $\prod(R_k)$ est la priorité de la tache de plus haute priorité accédant a R_k.

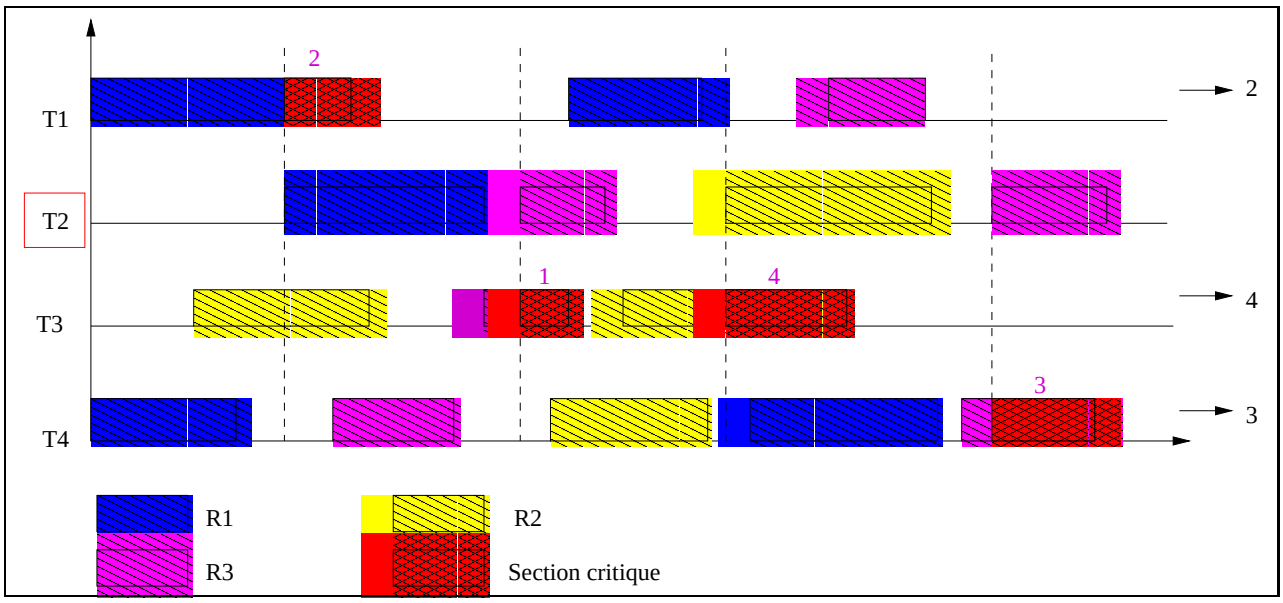


FIG. 3.8 – Temps de blocage avec PCP

Soit $P_1 < P_2 < P_3 < P_4$ (T1 est donc la tache la plus prioritaire). Considérons la tache T2 pour le calcul du temps de blocage.

Pour T_1 , $P_1 < P_2$ (T_1 est plus prioritaire que T_2) donc T_1 n'est pas a considérer dans l'algorithme.

Pour T_3 , $P_3 > P_2$ (T_2 est plus prioritaire que T_3). La première section critique rencontrée est sur R_3 et $\prod(R_3) \geq P_2$ (La tache de plus haute priorité accédant a R_3 est T_1 de priorité $P_1 \geq P_2$). La seconde section critique rencontrée est sur la ressource R_2 et $\prod(R_2) \geq P_2$ (La tache de plus haute priorité accédant a R_2 est T_2 or $P_2 = P_2$), il y a donc deux sections critiques a considérer (de 1 et 4 unités de temps).

Pour T_4 , $P_4 > P_2$ (T_2 est plus prioritaire que T_4). La seule section critique rencontrée est sur R_3 et encore une fois $\prod(R_3) \geq P_2$. Il y a donc une section critique a considérer de 3 unité de temps.

Une fois toutes les section critiques trouvées, la plus grande valeur est conservée, ici le temps de blocage de T_2 est de 4 unités de temps.

Les algorithmes ADA

```
1  function Pi_Rk (Resource_Name : in Unbounded_String;
2                  A_Resource_Set  : in Resources_Set;
3                  A_Task_Set      : in Tasks_Set
4                 ) return Priority_Range is
```

- Resource_Name : Nom de la ressource pour laquelle calculer le $\prod(R_k)$.
- A_Resource_Set : Ensemble des ressources du système.
- A_Task_Set : Ensemble des taches du système.
- retourne : $\prod(R_k)$.

Cette fonction permet de calculer le $\prod(R_k)$ d'une ressource k de nom Resource_name comme vu précédemment.

```
1 function Blocking_Time ( Task_Name           : in Unbounded_String;
2                          A_Task_Set          : in Tasks_Set;
3                          A_Resource_Set      : in Resources_Set
4                        ) return Natural is
```

- Task_Name : Nom de la tache pour laquelle calculer le blocking time.
- A_Task_Set : Ensemble des taches du système.
- A_Resource_Set : Ensemble des ressources du système.
- retourne : Le temps de blocage de la tache Task_Name.

Calcul le temps de blocage final pour une tache de nom Task_Name. Cette fonction utilise la fonction PCP_Critical_Tasks décrite ci-dessous.

```
1 function  PCP_Critical_Tasks ( Task_Name           : in Unbounded_String;
2                                A_Task_Set          : in Tasks_Set;
3                                A_Resource_Set      : in Resources_Set
4                              ) return Natural is
```

- Task_Name : Nom de la tache pour laquelle calculer le temps de blocage.
- A_Task_Set : Ensemble des taches du système.
- A_Resource_Set : Ensemble des ressources su système.
- retourne : retourne un premier résultat qui sera traité dans la fonction blocking_time.

Cette fonction permet de calculer un premier temps de blocage pour la tache Task_Name.

3.1.3 IHM de l'objet ressource

Choix technologique de la conception de l'IHM

Les IHM[1] nécessaires à l'objet ressource sont aux nombres de quatre, comme pour les autres objets existants (Processeur, Tâche) à savoir :

- L'interface d'ajout des ressources (figure 3.9).
- L'interface de modification des ressources (figure 3.10).
- L'interface de suppression des ressources.
- L'interface de visualisation des ressources existantes.

Les interfaces graphiques ont été décomposées en deux parties, une partie ne contenant que le code de la fenêtre et des appels aux fonctions du second fichier qui contient les actions associées aux boutons ou widgets.

```
-- nomfichier.adb (fichier1)
 Button_Callback.Connect (Update_Resources_Set.Button_Add, "clicked",
                          Button_Callback.To_Marshaller (On_Ok_Pressed'access));
-- nomfichier-callback.adb (fichier2)
 procedure On_Ok_Pressed ( Object : access Gtk_Button_Record'Class );
```

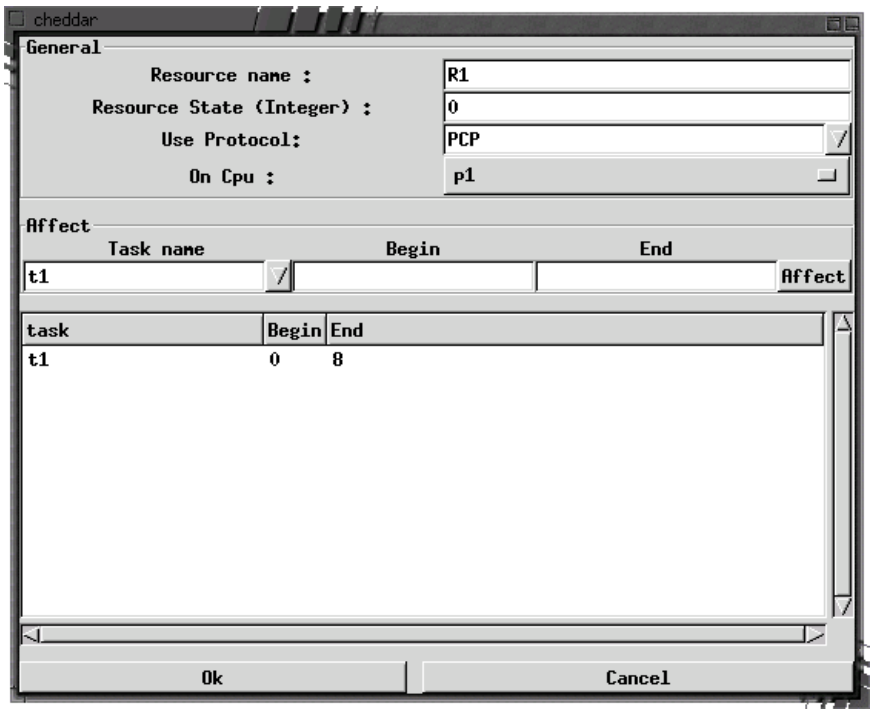


FIG. 3.9 – Interface d'ajout d'une ressource

L'interface (figure 3.9) d'ajout des ressources est d'une conception assez classique. Elle est composée de Widget Gentry pour les saisies au clavier, Une Combo_Box pour lister les protocoles pouvant être associés à la ressource, un Option_Menu pour la liste des processeurs existants et

[1] Interface Homme Machine

une CList pour afficher les tâches qui ont été associées avec la ressource. Dans cette interface l Option_Menu aurait put être remplacé par une Combo_Box ou inversement. Cependant bien que ces widgets semble réaliser les mêmes fonctions, il est impossible dans une Combo_Box de sélectionner un élément par défaut à afficher, alors que d'après la documentation GtkAda il est possible de le faire pour l'Option_Menu. Pour que l'utilisateur ne puisse pas entrer une tâche qui n'existe pas on à utiliser une Combo_Box qui regroupe les tâches existant dans le système.

Pour l'interface de modification des ressources, des filtres sur les processeurs et les ressources sont utilisés, leur but est d'éviter d'avoir une liste ou on retrouverait des ressources sans savoir à quelle processeur elles sont attachées, et une liste de tâche sans lien visible avec la ressource associée.

Lors du choix des éléments que l'utilisateur pourrait modifier, il a été décider d'interdire la modification du nom de la ressource, en effet le nom de la ressource est considéré comme étant un identifiant. Ce cas a était considéré différemment dans la gestion des tâches, ce qui peut conduire à une incohérence au niveau des tâches associées à une ressource, la tâche lors de la création porte un nom quelconque, mais son nom est modifié, donc la ressource à une tâche qui lui est associée qui n'existe plus, il est possible de régler le problème pendant la modification de la tâche, mais cette solution est très lourde en utilisation processeur si le nombre de ressources est important car il seras alors nécessaire de parcourir la liste des ressources puis celle des tâches associées à la ressource, on se retrouve donc dans le cas le plus défavorable à (Max_Ressources*Max_Tasks*Max_Processors) accès mémoires.

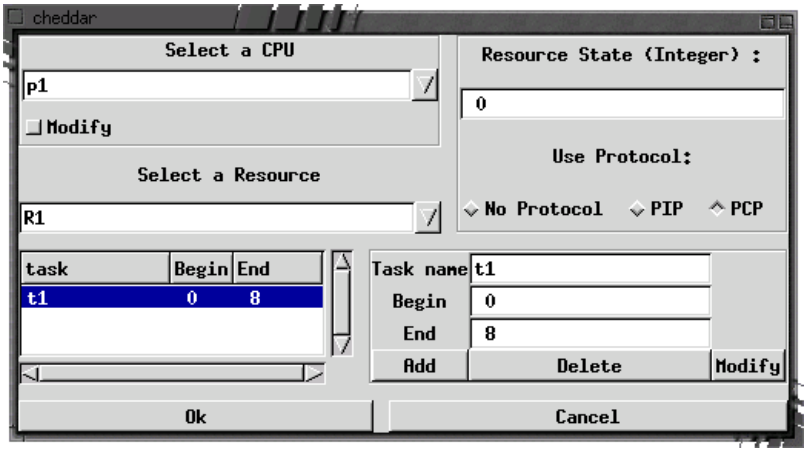


FIG. 3.10 – Interface de modification d'une ressource

Lors de la conception de l'interface de modifications des ressources (figure 3.10), plusieurs problèmes se sont posés lors de la conception de cette fenêtre, la première fut l'impossibilité de récupérer le signal émis par l Option_Menu de façon fiable, ce qui à été régler par son remplacement par une Combo_Box, ou on peut récupérer le signal lorsque le champ de la Gentry attaché à la Combo_Box est modifié. Ensuite le protocol de la ressource devait être modifié à chaque changement de ressource, un Option_Menu à donc été utilisé mais il c'est retrouvé que la fonction set_history du package Gtk_Option_Menu était buggé, bien que la chaine de caractères visible était bien celle voulue, le pointeur associé était toujours celui étant à la première position de la liste associé a l Option_Menu, on est donc passé à une Combo_Box mais aucune fonction n'était définie pour le faire, et créer cette fonction aurait été trop long, on est donc passé au Radio_Button qui nous à permis de palier au problème mais au détriment d'une évolution facile de la liste des protocoles

supportés par les ressources sans modification de l'IHM. Il a été demandé que la ressource puisse être changée de processeur, pour cela un Radio_Button a été rajouté, qui lorsqu'il est sélectionné fait apparaître une Combo_Box qui permettra de sélectionner le nouveau processeur, cette approche a été choisi pour éviter toute confusion du à la présence de deux Combo_Box contenant le même type d'information, lorsque le changement est pris en compte la Combo_Box est remasquée. Il est possible d'ajouter, de supprimer ou de modifier les tâches associées a une ressource, les changements effectués sont fait à chaque fois qu'un bouton est pressé, dans la limite ou le maximum de tâches pour une ressource n'est pas dépassé qui été fixé à 50.
Il existe trois méthodes pour que les changements soit pris en compte :

- En modifiant le processeur courant.
- En modifiant la ressource courante.
- En cliquant sur le bouton Ok.

Pour éviter qu'une sauvegarde soit effectué alors qu'il n'y as pas eu de modification, la valeur des champs est comparée aux valeurs se trouvant dans la structure de l'objet ressource, sans un nouveau parcours de la liste des ressources. Pour cela la fonction Resource_Modify a été définie pour être appelé par les différentes méthodes, cette fonction choisiras les paramètres de sauvegarde qui correspondent aux champs qui ont été modifiées.

```
--resource_Modify (update_ressource_pkg-callback.adb)
if Protocol /= Protocol_Resource
     or A_Resource.State /= State
     or Get_Active(Update_Resources_Set.Check_Change_CPU) = True
   then
     if Get_Active(Update_Resources_Set.Check_Change_CPU) = True
      then --Changement de processeur à la ressource
        CPU:=To_Unbounded_String(Get_Text(
                        Get_Entry(Update_Resources_Set.Change_CPU_List)));
        Set_Active(Update_Resources_Set.Check_Change_CPU, False);
        hide(Update_Resources_Set.Change_CPU_List);
        Update_Resource_CPU(Sys.Resources
                  , Update_Resources_Set.Last_Resource_Selected, CPU
                  , Protocol, State);
      else --Changements des paramétres
        if Update_Resources_Set.Last_CPU_Selected /= ""
          then
           if A_Resource.location=To_Unbounded_String(Get_Text(
                                Get_Entry(Update_Resources_Set.CPU)))
            then
              CPU:=To_Unbounded_String(Get_Text(
                               Get_Entry(Update_Resources_Set.CPU)));
              Update_Resource_CPU(Sys.Resources
                           , Update_Resources_Set.Last_Resource_Selected
                           , CPU
                           , Protocol, State);
            else
              Update_Resource_CPU(Sys.Resources
                           , Update_Resources_Set.Last_Resource_Selected
                           , A_Resource.location
                           , Protocol, State);
          end if;
       end if;
    end if;
end if;
```

Un mécanisme d'annulation à été mis en place, avant l'ouverture de la fenêtre l'objet ressource est sauvegardé, si l'utilisateur veut annuler les modifications effectuées, l'objet ressource est restauré, cette méthode ne permet pas de faire des annulations sur une partie des changements elle permet juste de revenir à un état initial, si une erreur de saisie c'est produite, il faudra remodifier la valeur du champs ou c'est produit l'erreur.

Le bouton OK valide les changements, il est alors impossible de modifier les changements fait auparavant.

3.2 Les contraintes de précédence

3.2.1 Le Canvas

Le canvas est une vue de Cheddar qui n'affiche que les systèmes d'objets possédant des précédences. Tout autre système est caché dans cette vue. Elle sert à représenter les objets créés dans Cheddar de manière graphique. Le but initial de cette fenêtre est de proposer une représentation sous la forme d'un graphe composé d'entités et d'arcs pour les relier. De plus, le graphe ne doit pas être coercitif (pas de circuit).

La fenêtre doit aussi être pourvue d'une barre d'outils comprenant des boutons qui donnent accès aux fonctionnalités du canvas.

La Barre d'outils

Choix des objets
Plusieurs choix d'objets et de disposition se présentent pour la barre d'outils. En effet, étant donné le nombre d'objets disponibles, il existe plusieurs combinaisons pour chacun des champs nécessaires et un choix est donc obligatoire en fonction des contraintes requises et des fonctionnalités de chaque objet.

Les différents objets de Gtk existant et compatibles avec les besoins de l'interface sont :
- combo box
- option menu
- buttons
- radio buttons
- check box
- pixmap
- entry
- toggle buttons
- clist

Pendant la construction de cette toolbar diverses combinaisons ont été essayées pour arriver à la solution de deux toolbars placées l'une sous l'autre dans une Vbox. La première toolbar est composée d'une combo box qui permet de choisir le mode d'utilisation du canvas, les modes présents sont création, sélection et suppression d'item.

Viennent ensuite les fonctions de zoom respectivement zoom out et zoom in puis les différentes dispositions, droit, cercle et flux. L'icone suivante permet de fermer le canvas.

La seconde toolbar est composée de radio buttons qui permettent de choisir l'objet sur lequel va agir le mode sélectionné. Ces radio buttons représentent respectivement les objets dépendance, tache, message et buffer.

Raisons de ces choix
L'affichage sélectionné pour le mode a donc été la combo box car elle offre une meilleure lisibilité pour l'utilisateur et possède les caractéristiques nécessaires à ce champ. Le choix des radio buttons s'explique par le fait que cela hiérarchise les disponibilités pour chaque mode. L'utilisation de pixmap pour les autres objets vient de la volonté d'homogénéité avec les autres fenêtres de Cheddar.

La zone de dessin

Afin de mettre en oeuvre la zone de dessin, un widget de GtkAda a été très utilisé : le canvas (widget `GtkAdA.Canvas`). Ce widget propose un ensemble de fonctions de haut niveau permettant à l'utilisateur de placer des objets, de les bouger avec la souris,...Les items peuvent être connectés entre eux et les connections restent actives même si les items sont bougés. Ce widget supporte aussi les barres de défilement (`Gtk_Scrolled_Window`) qui apparaissent quand le dessin dépasse les dimensions de la fenêtre.

Hélas, certaines méthodes ou fonctions manquent dans ce widget :

- la liste des items
- le nom d'un item
- la recherche par nom
- l'item sélectionné (pour traçage de flèches ultérieur)
- le type des items (taches, messages, tampons)

Le code source du widget Canvas a donc été repris (sous le nom de `my_canvas`) et les fonctionnalités manquantes ont été rajoutées.

Fonctionnement général du canvas

La vue 'graphes' peut fonctionner en parallèle avec Cheddar, c'est-à-dire que l'on peut créer des tâches dans Cheddar et l'instant d'après les afficher dans le canvas.

Création d'objets

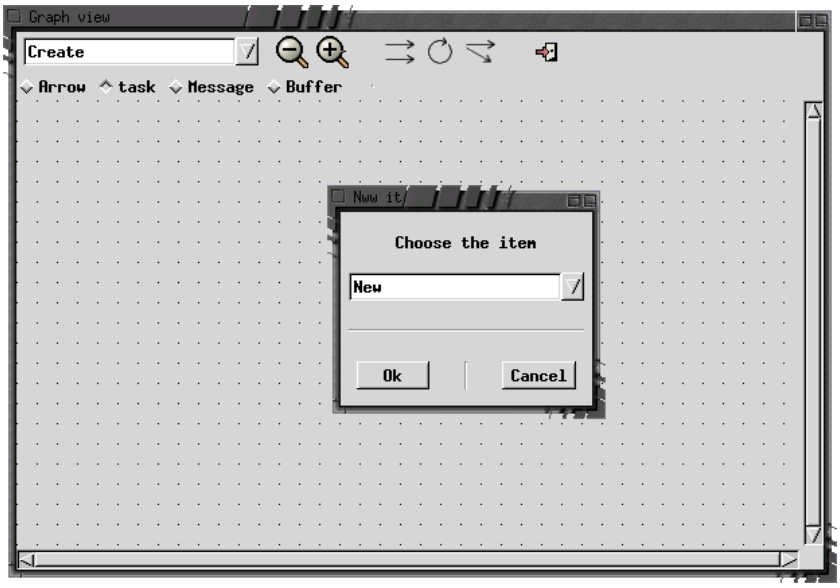


FIG. 3.11 – Le Canvas

Quand l'utilisateur désire créer un message, il passe en mode création, il sélectionne le type d'item qu'il veut créer dans la barre d'outils et il clique quelque part sur le canvas. A ce moment, une fenêtre popup apparaît et propose à l'utilisateur de :

- créer un nouveau message. Dans ce cas, la fenêtre de création de message apparaît, l'utilisateur renseigne les caractéristiques du message, valide (en appuyant sur Ok) et l'item est affiché sur le canvas.
- d'importer un message de Cheddar. Ainsi, l'item est simplement affiché sur le canvas.

Afin de différencier les items sur le canvas, leur nom est affiché à proximité de chacun. L'utilisateur peut zoomer à loisir sur le canvas. De plus, 3 types de disposition ont été mis en place. Hélas, les algorithmes utilisés pour ces positionnements sont à revoir car il ne prennent pas en compte les dépendances et les contraintes de précédence.

Relations entre items

Les items peuvent être reliés entre eux via des flèches qui sont en fait des courbes de Bezier. Ceci permet au canvas de gérer les liens multiples entre items. Ainsi, quand plusieurs liens sont créés entre deux items, les flèches ne sont pas superposées les unes sur les autres mais étalées en courbes.

Suppression d'objets

Aucun item n'est réellement supprimé du canvas, mais caché. Cela a plusieurs avantages :
- amélioration des performances générales
- conservation de la cohérence des données avec les objets du système Cheddar

En effet, l'utilité première du canvas est d'afficher et/ou de créer des objets Cheddar, mais pas de les supprimer. Les items ainsi cachés peuvent être réaffichés par le biais de la procédure de création d'items.

Quand un item est ainsi caché, les liens partant et venant de lui sont aussi cachés.

Beaucoup de problèmes sont apparus au cours du développement de la vue 'graphes' :

1. Concernant la barre d'outils, un certain nombre de versions ont été nécessaires avant d'arriver à la version finale qui est simple d'utilisation et facile à comprendre.
2. Le canvas a été dur à implémenter car :
   - le mécanisme de création et d'affichage d'items n'est pas évident à assimiler.
   - beaucoup de relations existent entre le système d'objets de Cheddar et la vue 'graphes'.
   -

Par la suite, le mécanisme des contraintes de précédence va être expliqué.

3.2.2 Contraintes de précédence

Explications théoriques

Les dépendances n'ont de sens qu'entre deux tâches. Ces deux tâches sont alors liées par une contrainte de précédence. Les dépendances peuvent être de trois types différents :

```
type Dependency_Type is
       (Precedence_Dependency,
        Buffer_Dependency,
        Communication_Dependency);
```

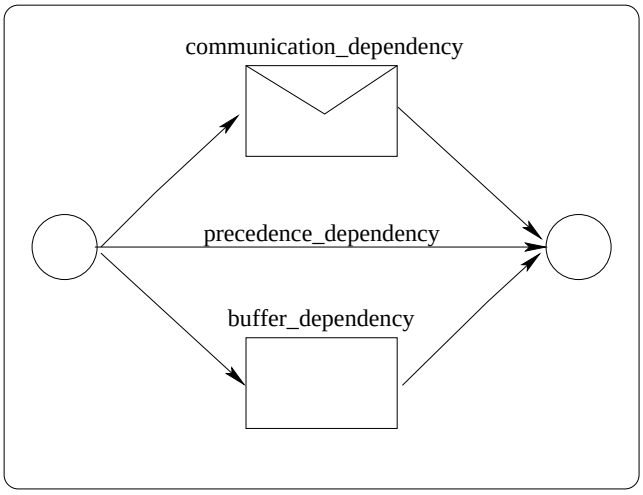


FIG. 3.12 – types de dépendance

La Première est une simple contrainte de précédence entre deux tâches. La seconde inclue un tampon entre les deux tâches, ce tampon contient des règles. Le tampon est d'arité n-n.La troisième inclue un message entre les deux tâches, ce message contrairement au tampon est d'arité 1-1, il ne concerne donc que deux tâches.

Ces dépendances sont enregistrées dans une matrice. Cette matrice est composée de depends_set : un set de pointeurs de dépendance. Chaque ligne ou colonne est associée à une tâche.

```
type Depends_Set is new Dep_Set.Set with null record;
...
type Tasks_Table is array (Dependencies_Range) of Task_Ptr;
type Dependencies_Table is array (Dependencies_Range,
                                  Dependencies_Range) of Depends_Set;
```

Le packet Set est un packet générique. Le type Depends_Set dépend de ce packet, il utilisera toutes les procédures et méthodes du package.

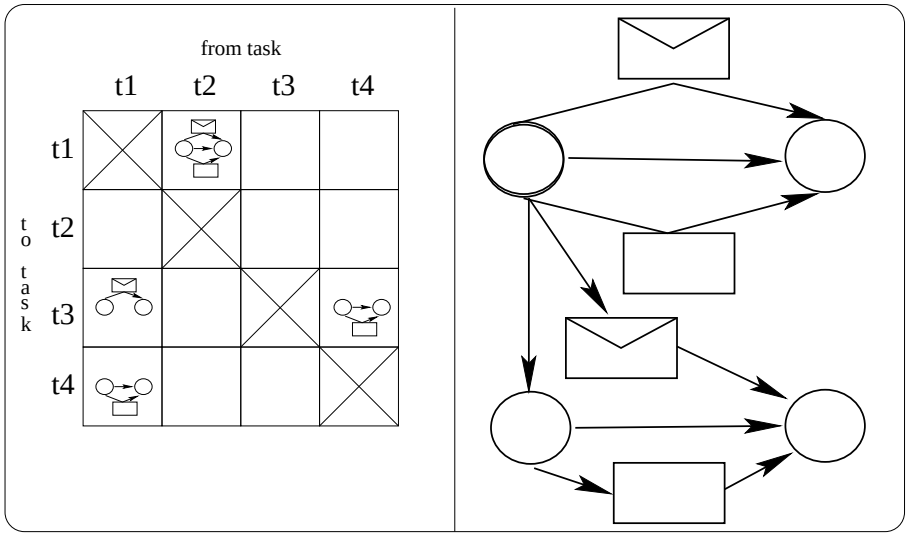


FIG. 3.13 – représentation dans la matrice

La structure *Task_dependencies* permet de maintenir les informations liées aux dépendances.

```
type Tasks_Dependencies is
    record
       Dependencies    : Dependencies_Table;
       Task_Position   : Tasks_Table;
       Number_Of_Tasks : Dependencies_Range := 0;
    end record;
```

Une variable dependencies est inclue dans la structure du packet system. Cette variable est accessible a l'aide de la variable publique sys par la notation pointée : *sys.dependencies*.

Le tableau Task_Position permet d'associer une position à une tâche. La position dans la matrice d'une tâche est retrouvée à l'aide de l'indice de la tâche dans ce tableau.

Malheureusement un problème subsiste. Lors de la suppression d'une tâche les dépendances de celle-ci sont bien supprimées mais elle n'est pas rayée de la matrice. La tâche est toujours présente dans le tableau Task_Position et donc dans la matrice des dépendances. Le problème ayant été repéré trop tard, il n'est pas réglé.

Ce problème peut être résolu à l'aide d'une méthode dans task_dependencies mettant a jour la matrice et le set des tâches.

Mise en place dans cheddar

l'implémentation des dépendances était déjà présentes ainsi que celle des messages et des tampons. Ces structures sont utilisées par le canvas et permet de maintenir les informations relatives aux dépendances. Quelques modifications minimes ont due être effectuées :

- suppression de dépendances
- modification de méthode de test

L'utilisation d'un package, graph_dependencies, permet d'effectuer le lien entre le canvas et la structure de Cheddar. Ce package traduit les différentes informations récoltées par le canvas et lance l'enregistrement interne des modifications. Il oriente les modifications en fonction de l'état du graph présent sur le canvas.

L'enregistrement interne consiste a conserver les dépendances crées par l'utilisateur a l'aide du canvas. Ces dépendance sont maintenues dans la matrice décrite ci-dessus.

Afin de conserver toutes les informations liées aux dépendances, le type half_dep est nécessaire :

```
type Half_Dep_Type is
      (To_Task,
       To_Dependency);
```

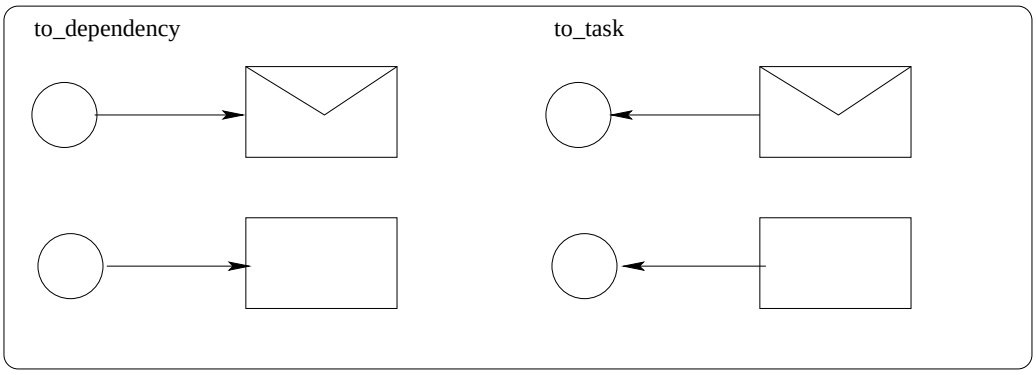


FIG. 3.14 – types de semi_dépendence

```
type Half_Dep is record
     The_task : Task_Ptr;
     The_Dep  : Dependency_Ptr;
     The_Type : Half_Dep_type;
end record;
```

les "semi-dépendences" c'est à dire les liens tâche-tampon ou tâche-message permettent un maintient plus fin de l'information liée a une dépendence.

Le premier essai n'utilisait pas ces semi_dépendences mais comportait certaine contraintes d'utilisation. Par exemple, l'utilisateur ne pouvait définir de lien buffer-tâche ou message-tâche que si auparavant il avait saisi une semi-dépendences entre une tâche et ce même buffer ou message.

Grâce a l'usage de cette structure ce problème n'est plus. Toutes les configurations du graph sont enregistrées et peuvent être modifiées sans perte d'information.

les couples de semi_dépendence forme une dépendence. C'est pourquoi lors de la création d'une semi_dépendence, la vérification de l'existence de sa pair est nécessaire. Ainsi si celle-ci existe, la dépendence correspondante est crée. Bien sûr, il en va de même pour la suppression.

Une fois la structure mise en place, le débuggage effectué, la liaison avec les évènements de cheddar restait à faire. Bien entendu lors de l'ouverture d'un nouveau projet ou d'une mise à jour, le canvas est reconstruit. Plusieurs cas de figures sont alors possibles :

- soit le projet n'est pas le même

- soit les modifications ont été effectuées sur le projet courant

lorsque le projet est le même, au niveau des dépendences il suffit de prendre en compte les modifications. La solution retenue fût de placer des appels de procedure dans les callbacks concernés. Par exemple, la suppression d'un tâche entraînera la suppression de toute ses dépendences.

Un changement de projet entraîne la reconstruction des différentes variables de la structure du système. Pour mettre à jour les semi_dépendences deux essais ont été effectués :

- La première idée fût de mettre en place une procédure permettant de synchroniser les variables *dependencies* et *half_dependencies* de la structure du système. Cette procédure parcoure le set des dépendences et ajoute les semi_dépendences correspondantes dans la variable *half_dependencies*. Cette méthode contenait plusieurs boucle pour que les informations soient cohérentes et pour parcourir le set des dépendences.
- La seconde solution permet d'éviter l'ajout de parcourt de set. L'ajout de semi_dépendences ce fait lors de la construction du set des dépendences. Ainsi l'algorithme est moins gourmand en ressource. Pour chaque dépendence, deux semi_dépendences sont ajoutées. Cependant cette méthode dans le cas du chargement d'un buffer ayant plusieurs destinataire et plusieurs sources, l'exception$EXIST_ALREADY$ peut se lever. Le problème peut être réglé en modifiant la méthode *read_from_file* du fichier *task_dependencies.adb*. A chaque ajout d'une semi_dépendences, il faut gérer les exceptions car sinon ce type de projet ne peut être chargée. Un doute subsiste, L'exception n'aie jamais apparue lors de nos test. Ce cas est donc à tester plus précisément.

Illustration par un exemple

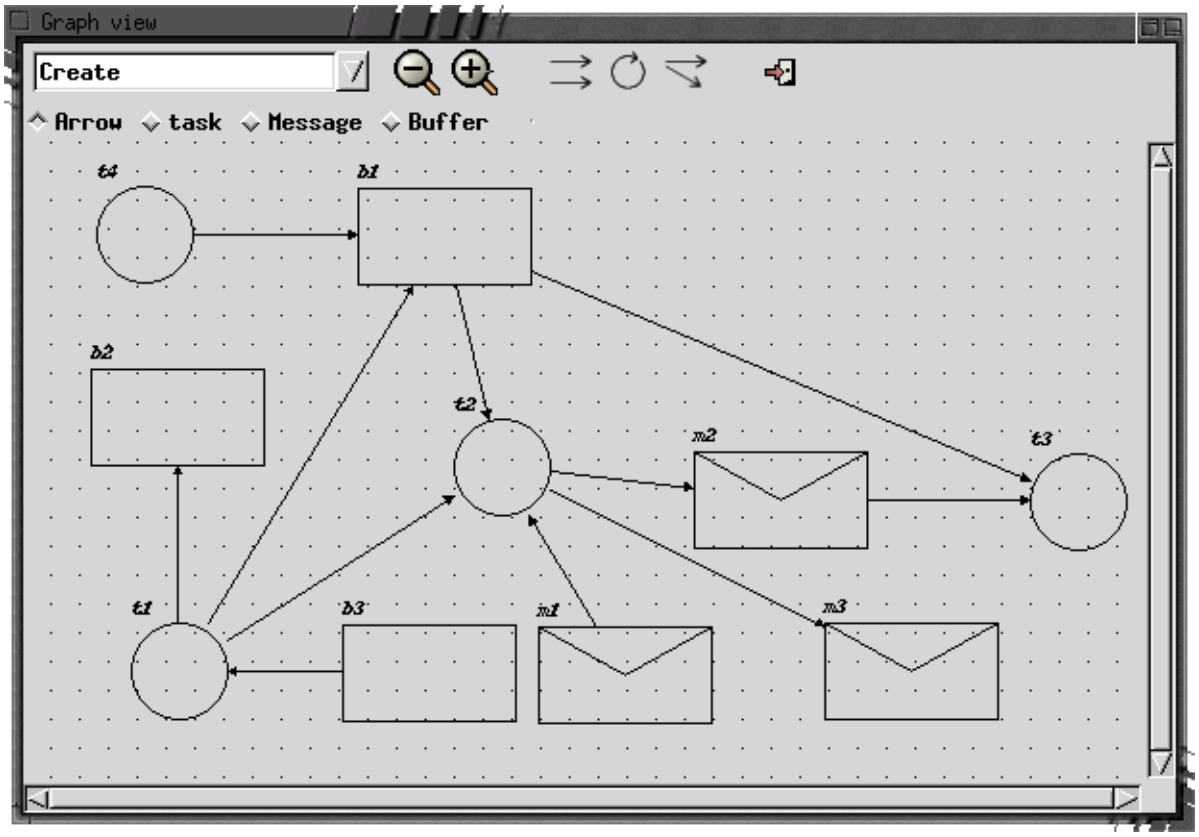


FIG. 3.15 – exemple cheddar

Cet exemple comporte plusieurs cas de figure illustrant les dépendences et semi_dépendences.

La tâche t4 ne possède qu'une semi_dépendence vers le tampon b1. Cette tâche n'a donc pas de dépendence enregistrée. Cependant ,si le canvas est rechargé pour une raison quelconque, cette tâche restera apparente.

La tâche t1 possède une précédence avec la tâche t2, cette dépendence n'est pas enregistrée comme semi_dépendence. Les buffers b2 et b3 ont une semi_dépendence avec t1. Ces semi_dépendences sont maintenues dans le set `halph_depends`. Ce set fait parti de la structure système.

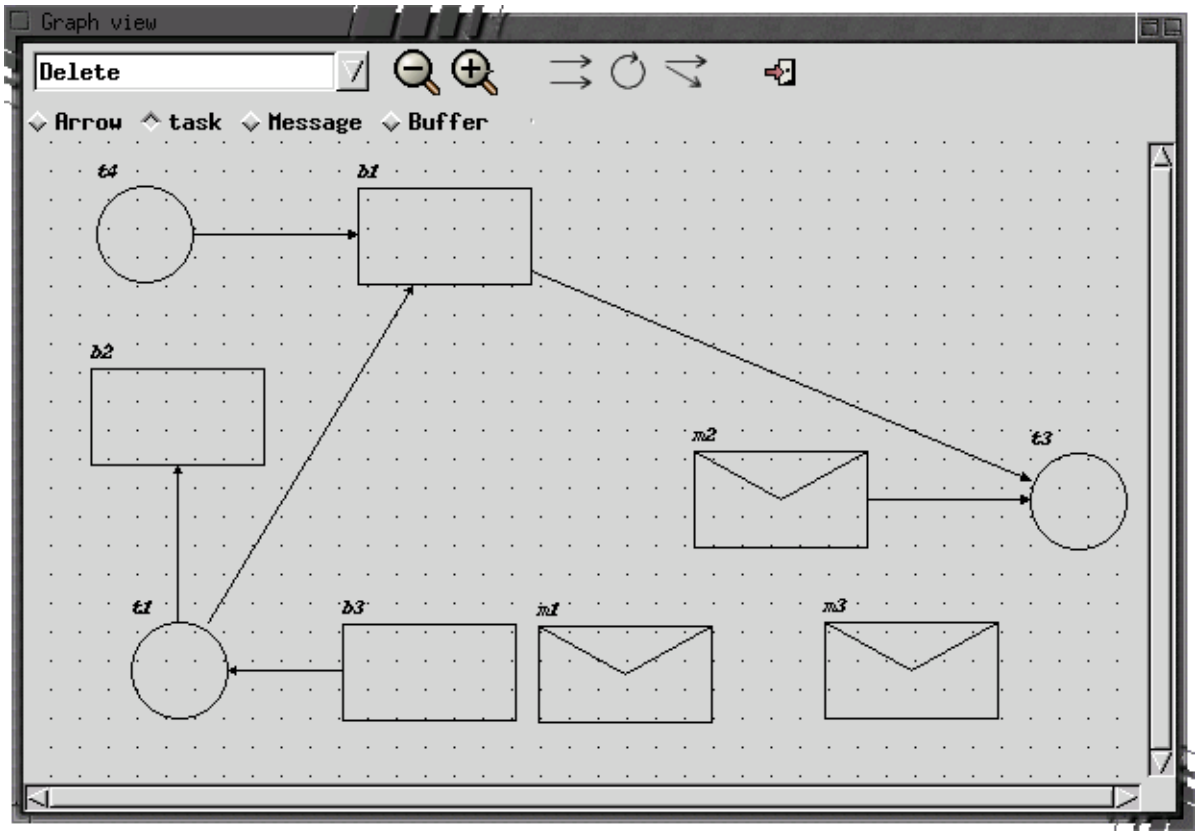


FIG. 3.16 – Suppression de t2

La tâche t2 est supprimée. Elle n'est en fait pas supprimée au premier sens du terme, elle est simplement cachée. En effet une suppression dans le canvas équivaut a supprimer toutes les dépendences et semi_dépendences.

Le canvas étant une vue d'un système possédant des dépendences, la tâche t2 n'a plus de raison d'être visible. Elle est donc rendue non visible. Les messages m1 et m2 n'ont plus de dépendences et pourraient être cachés à leur tour. Cependant ces messages pouvant servir ultérieurement avec d'autre tâche ils restent visibles.

L'utilisateur peut aussi supprimer messages et buffers de la même façon. Les dépendences et semi_dépendances liées a ces items seront alors supprimées.

3.2.3 IHM (Taches, Messages, Tampons)

Les objets Cheddar sont stockés dans des `sets` Ada. Ces sets sont regroupés dans un record :

```
type System is tagged
 record
    Network      : Network_Ptr;
    Tasks        : Tasks_Set;
    Resources    : Resources_set;
    Messages     : Messages_Set;
    Dependencies : Tasks_Dependencies;
    Half_Dependencies : Half_Depends_Set;
    Processors   : Processors_Set;
    Buffs        : Buffers_Set;
 end record;
```

Une variable globale de type System est utilisée partout dans Cheddar, elle s'appelle `sys`. Il y a un set par type d'entités. Ainsi, par exemple, les tâches sont stockées dans `sys.tasks`.

Le contenu de chacun de ces sets est géré via des IHM spécifiques. La construction des IHM Tâche, Message et Tampons est détaillée par la suite.

Modification de l'IHM tache

Raisons

Une IHM pour les taches était déjà présente dans cheddar, cependant étant donné que l'identifiant d'une tache est son nom, l'implémentation des dépendences a rendu le procédé existant obsolète. Dans l'ancienne version de l'update pour mettre à jour une tache, toutes les taches étaient effacées puis reconstruites avec leurs nouveaux paramètres. Une procédure `Update_Task` a donc été créée dans le package `task_set` de façon a rendre l'update compatible avec l'implémentation de dépendances, cette procédure modifie directement les champs de la tache a mettre à jour dans `sys.tasks`, le set des taches du système. L'IHM de mise à jour a donc du être aussi modifié pour la rendre compatible au nouveau type d'update et plus fonctionnelle.

Modifications

Pour l'implémentation des dépendences la procédure `Update_Task` a donc été ajoutée, son prototype est le suivant :

```
procedure Update_Task (
         My_Tasks      : in out Tasks_Set;
         Name          : in     Unbounded_String;
         New_Name      : in     Unbounded_String;
         Location      : in     Unbounded_String;
         Start_Time    : in     Natural;
         Capacity      : in     Natural;
         Period        : in     Natural;
         Deadline      : in     Natural;
         Jitter        : in     Natural;
         Blocking_Time : in     Natural;
         Priority      : in     Priority_Range;
         Offset        :        Offset_Table );
```

L'ancienne version était faite de `Gtk_entry`, chaque ligne représentant une tache et chaque colonne un paramètre de cette tache. Sur cette version `sys.tasks` était vidé puis réinitialisé avec les données de chaque ligne. Cette réinitialisation supprimait les liens des taches vers leurs dépendances. Pour éviter cela la nouvelle version modifie les paramètres existant de la tache à modifié au lieu de les supprimer puis les recréer. Dans la nouvelle version les taches existantes apparaissent dans une `clist` avec leurs paramètres. En cliquant sur une tache de la liste ses paramètres remplissent les champs a gauche et il est donc ensuite possible de modifier ceux-ci. Les modifications ne sont prises en compte que lorsque l'on appuie sur le bouton modify. Cette action remplace tous les champs de la tache, qui est identifiée par son nom, par les valeurs des champs de gauche de l'interface.

Ci-dessous la capture d'écran de l'interface :

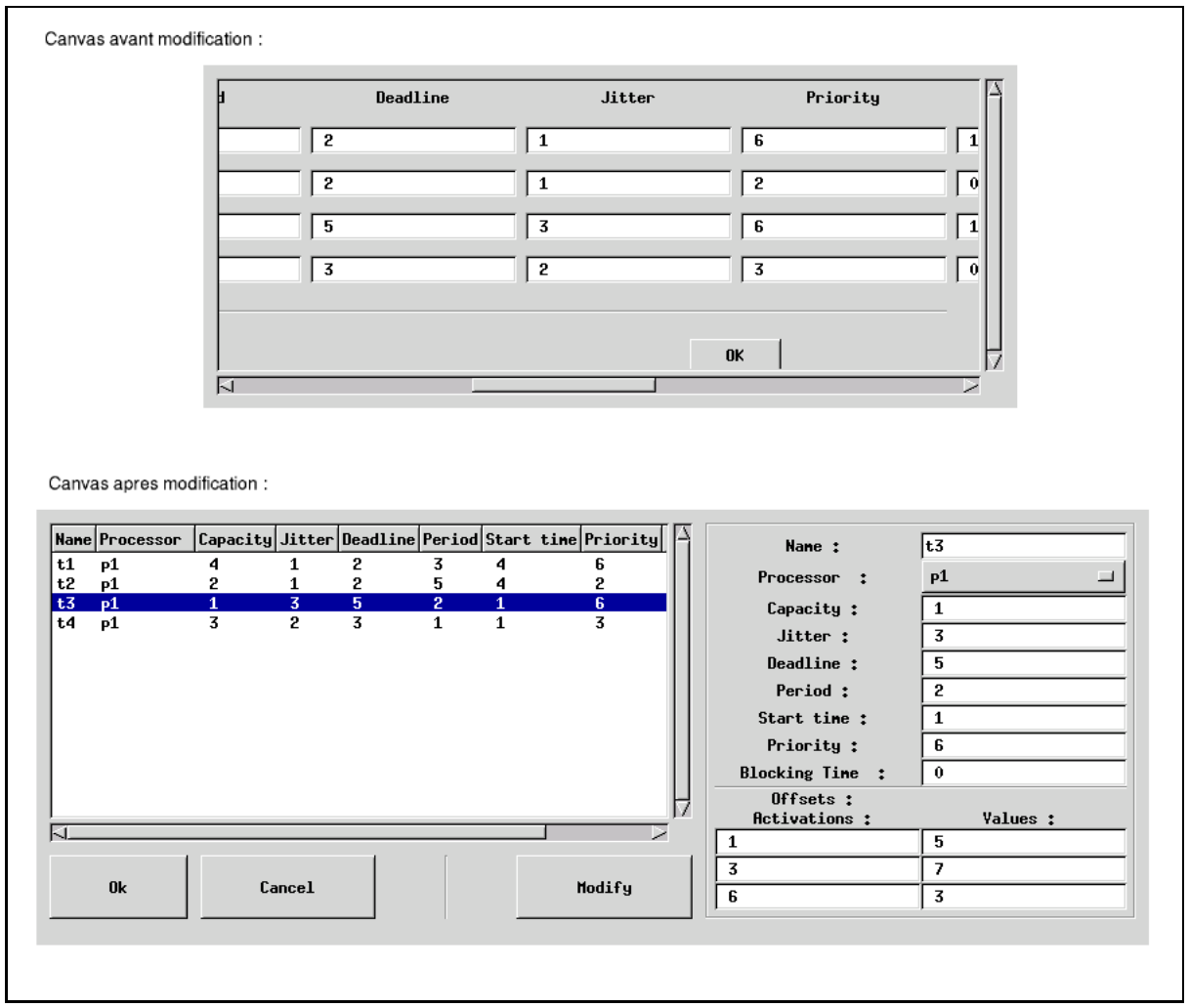


FIG. 3.17 – Comparaison entre l'ancienne et la nouvelle IHM Tache

Dans cet exemple, quatre tâches ont été créées et la tache *t3* sélectionnée. Ses paramètres sont affichés dans les champs du cadre de gauche et chacun d'entre eux peut être mis à jour par édition du champs correspondant à part le champs `blocking time` qui ne dépend que du protocole choisi pour la ressource à laquelle la tâche est affectée si c'est le cas et est égal à zéro sinon.

Lors d'une mise à jour du nom d'une tâche ou de plusieurs de ses champs, `sys.tasks` est mis jour avec les nouveaux paramètres de la tâche identifiée à gauche. La liste de droite est ensuite mise jour par rapport à `sys.tasks`. Le canvas est directement mis à jour après une modification et les noms de tache(s) modifiée(s) s'il y a eu mise à jour.

IHM message

Un message est un type de dépendance servant de communication entre différentes tâches, en plus de la fin de la tâche précédente cela permet d'ajouter une contrainte d'attente d'un message spécifique. L'IHM pour les messages n'existait pas, la création d'interface création, mise à jour et suppression a donc été implémentée pour `sys.messages`, le set des messages du système.

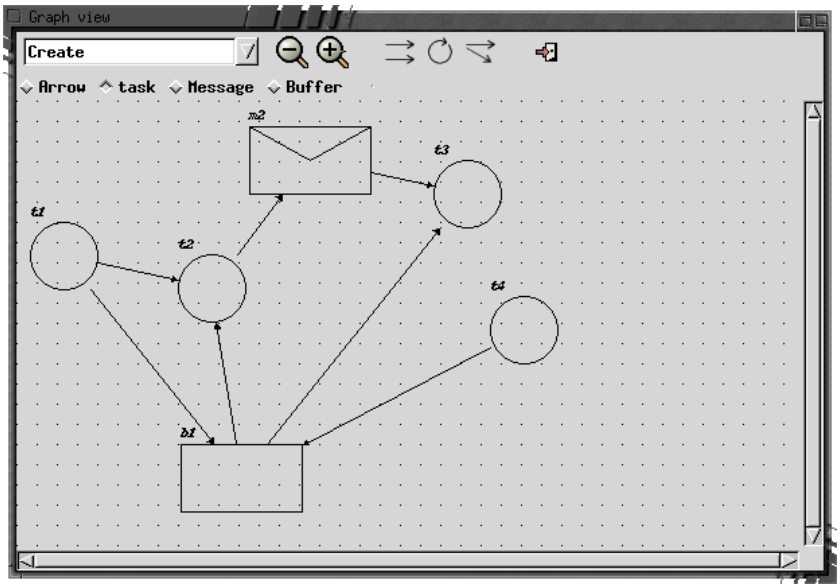


FIG. 3.18 – Représentation dans le canvas

Ajout d'un message

L'interface ci-dessous est la création d'un message :

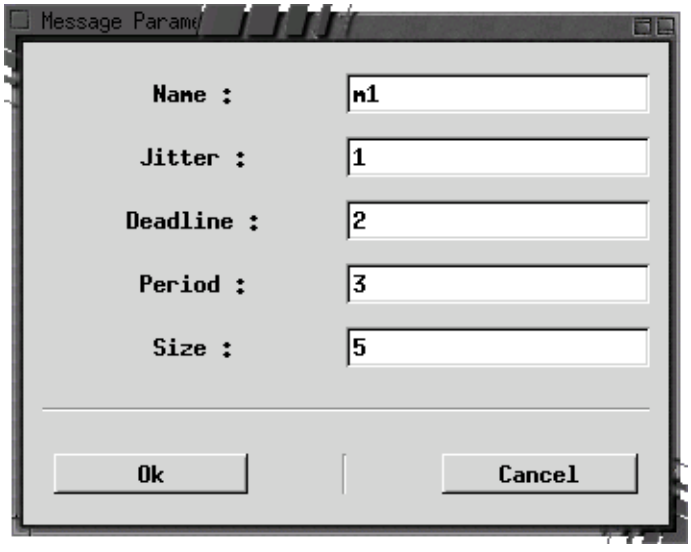


FIG. 3.19 – interface de création d'un message

Mise à jour d'un message

L'interface ci-dessous est la mise à jour des message :

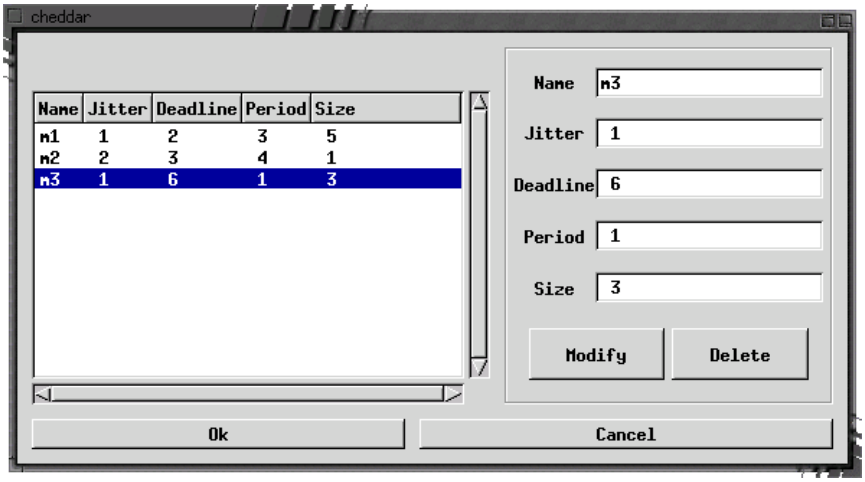


FIG. 3.20 – interface de mise à jour des messages

La procédure `Update_Message` a donc été ajoutée, son prototype est le suivant :

```
procedure Update_Message (
       My_Messages : in out Messages_Set;
       Name        : in     Unbounded_String;
       New_Name    : in     Unbounded_String;
       Size        : in     Natural;
       Period      : in     Natural;
       Deadline    : in     Natural;
       Jitter      : in     Natural );
```

Cette procédure modifie les paramètres existant du message à modifier, les messages existant apparaissent dans une clist avec leurs paramètres. En cliquant sur un message de la liste ses paramètres remplissent les champs à gauche et il est donc ensuite possible de modifier ceux-ci. Les modifications ne sont prises en compte que lorsque l'on appuie sur le bouton modify. Cette action remplace tous tous les champs de la tache, qui est identifié par son nom, par les valeurs des champs de gauche de l'interface. Il est aussi possible de supprimer un message par le bouton delete, la tache supprimée est celle sélectionnée dans la clist.

Sur la figure 3.20, trois messages ont été créés et le message *m3* sélectionné. Ces paramètres sont affichés dans les champs du cadre de gauche et chacun d'entre eux peut être mis à jour par édition du champ correspondant.

Lors d'une mise à jour du nom du message ou de plusieurs de ses champs, `sys.messages` est mis jour avec les nouveaux paramètres du message identifié à gauche. La liste de droite est ensuite mise jour par rapport à `sys.messages`. Le canvas est directement mis à jour après une modification et les noms de message(s) modifiée(s) s'il y a eu mise à jour.

IHM de Buffer

Un Tampon est une dépendance d'arité n-n. C'est-à-dire que un tampon peut être relié à plusieurs tâches. Un Buffer permet de stocker des informations temporaires concernant des tâches. Il peut accueillir des informations sur plusieurs tâches. Il faut aussi spécifier au buffer le type d'accès aux données (Consommateur ou Producteur). La réservation des données pour une tâche se fait par l'intermédiaire des Rôles. Un buffer est limité par sa taille et la taille totale des rôles doit être inférieure ou égale à celle-ci.

La structure de données Buffer

Un Buffer est un objet dont la structure est la suivante :

```
type Buffer is new Dependency with
 record
    Name     : Unbounded_String   := To_Unbounded_String ("");
    Location : Unbounded_String   := To_Unbounded_String ("");
    Size     : Natural            := 0;
    Rules    : Buffer_Rules_Table;
 end record;

type Buffer_Ptr is access all Buffer'Class;
```

Un Buffer est un objet identifié par son nom. Il est localisé sur un processeur, a une taille prédéfinie et un ensemble de Rôles. Le Type `Buffer_Rules_Table` est en fait un tableau d'objets de type `Buffer_Rule` :

```
type Rule is
      (Norule,
       Producer,
       Consumer);
type Buffer_Rule is
   record
      The_Rule : Rule;
      Size     : Natural;
   end record;
```

L'ajout de Tampon

L'utilisateur peut ajouter des tampons au système uniquement après avoir défini un processeur et une tâche. Un buffer est relié à un processeur, a une taille et un identifiant (son nom). Un buffer peut contenir plusieurs Rôles. Chaque rôle est associé à une tâche, a une taille et est de type Producteur ou Consommateur.

Cette fenêtre est basé sur l'IHM d'ajout de ressource (figure 3.9). Ainsi, une certaine homogénéité est assurée entre les deux types d'objets.

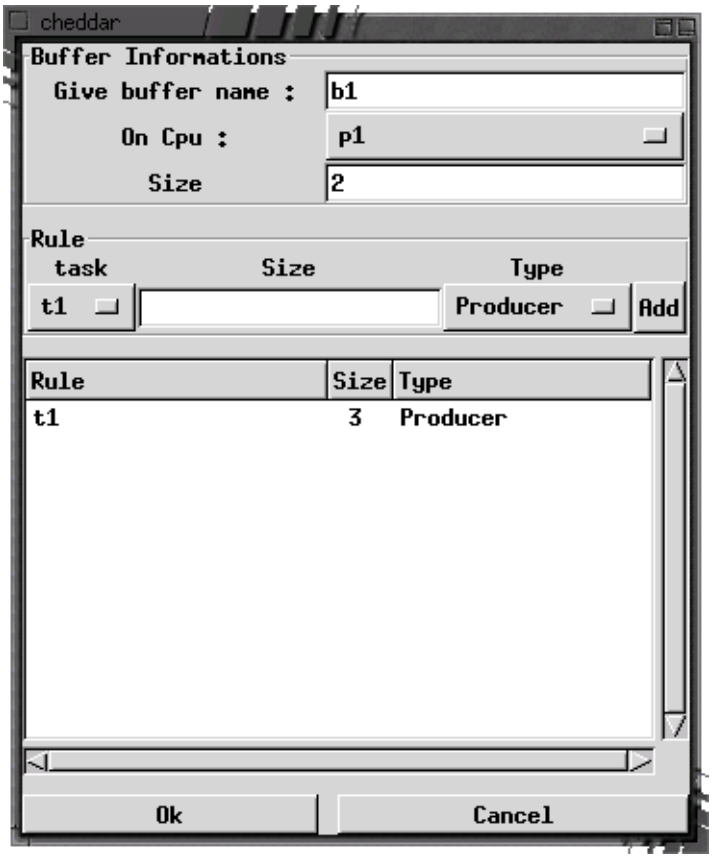


FIG. 3.21 – Ajout d'un tampon

Cette fenêtre contient un certain nombre de menus générés grâce aux informations (tâches, processeurs) de Cheddar. Pour la gestion de ces données le widget `Gtk_Option_Menu` a été préféré pour son aspect graphique et sa programmation facile.

La mise à jour de Tampon(s)

Initialement, la mise à jour d'un tampon se fait en le supprimant et un autre tampon (ayant les mêmes caractéristiques) est créé. La procédure `Update_Buffer` a donc été réécrite de la même manière que `Update_Message` (cf section 3.2.3) et a le prototype suivant :

```
procedure Update_Buffer (
    My_Buffers : in out Buffers_Set;
    Name       : in     Unbounded_String;
    New_Name   : in     Unbounded_String;
    Size       : in     Natural;
    Location   : in     Unbounded_String;
    Rules      : in     Buffer_Rules_Table );
```

Cette procédure met à jour le tampon (`Name`) dans un set de tampons (`My_Buffers`) avec les nouvelles propriétés (`New_Name`,`Size`, `Location` et `Rules`).

L'IHM de mise à jour de tampons fonctionne de la manière suivante :

1. il faut d'abord sélectionner le buffer à modifier via un combo box. Une fois cela fait, les propriétés du combo apparaissent à droite du combo. Ces informations sont modifiables directement et validables via un bouton 'Modifier'.
2. une fois le tampon sélectionné, ses rôles apparaissent dans une clist. Pour modifier un rôle, il suffit de cliquer dessus dans la clist et ses propriétés sont affichées dans des champs modifiables. Il est de plus possible d'ajouter ou de supprimer des rôles pour un tampon donné.

Ci-dessous, une capture d'écran de la fenêtre de mise à jour de tampon :

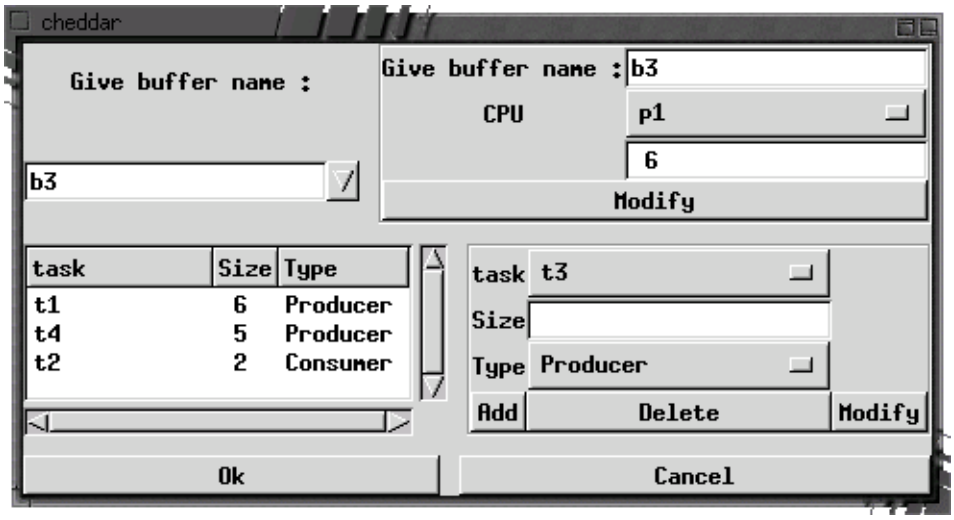


FIG. 3.22 – Modication de tampon

Lors d'une mise à jour de tampon(s), le set `sys.buffers` est modifié en conséquence. Le canvas est aussi mis à jour. Ainsi, la cohérence des données est conservée.

3.3 Manuel d'utilisation

3.3.1 blah

3.3.2 Le canvas

Pour ouvrir le canvas, cliquez sur l'icone :

FIG. 3.23 – icône du canvas

la fenêtre canvas est alors ouverte. Si le projet

3.3.3 IHM Ressource

Ajout d'une ressource

On appel la fonction d'ajout de ressource via le menu : Edit -¿ Add -¿ Add Resource

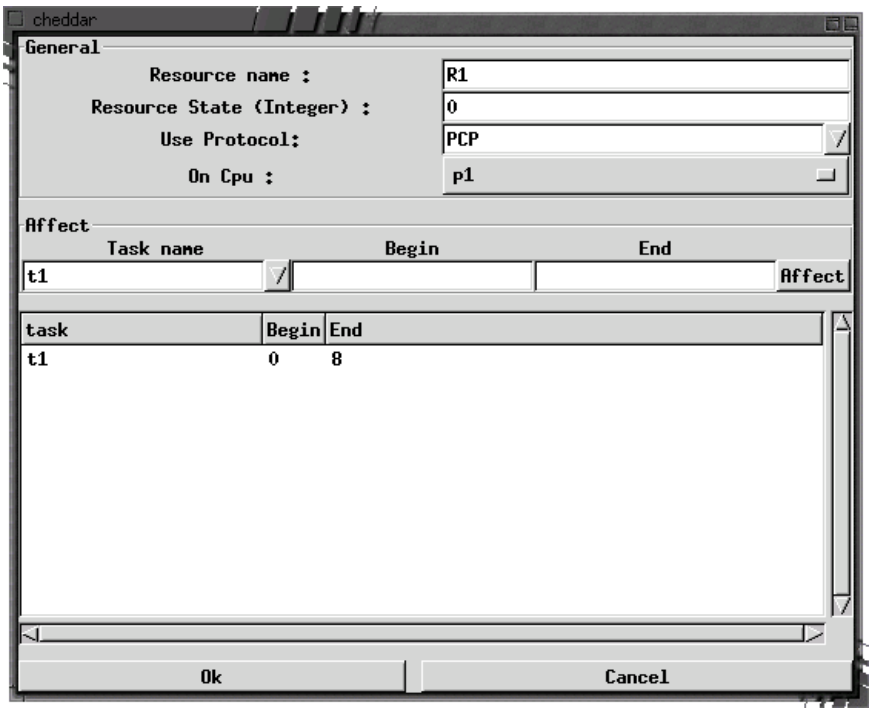


FIG. 3.24 – Interface de modification d'une ressource

Prérequis à la création d'une ressource :
- Un processeur doit exister.
- Une tâche doit exister.

Le nom de la ressource doit être ensuite spécifier, ainsi que son état initial, on doit ensuite spécifier le protocole partage de ressource (PCP[2] , PIP[3], No protocol), et le processeur sur lequel

[2]Priority Ceiling Protocol
[3]Priority Inhérence Protocol

sera attaché la ressource. On peut aussi lui attacher des tâches, il faut alors sélectionner la tâche que l'on veut parmi celle qui sont créées et de spécifier le début et la fin de son accès à la ressource Et de cliquer sur affect pour l'ajouter à la liste et cela autant de fois que nécessaire dans la limite fixé pendant la compilation. une fois fini il suffit de cliquer sur Ok pour valider la création ou cancel pour annuler.

3.3.4 Modification d'une ressource

On appel la fonction de modification via le menu : Edit -¿ Update -¿ Update Resource

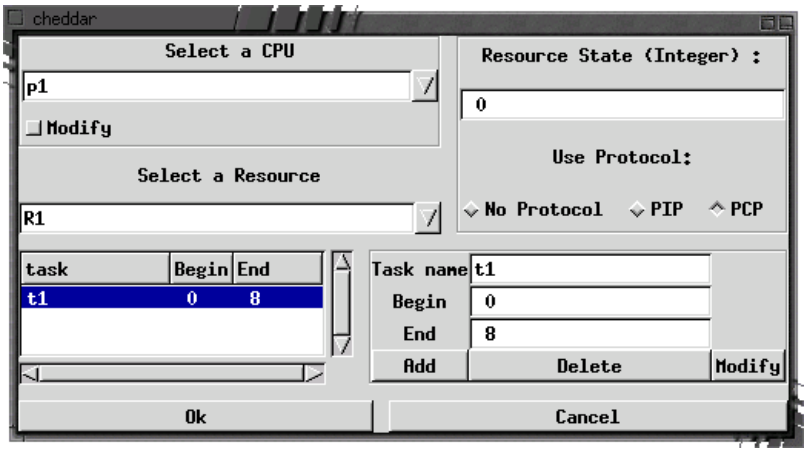


FIG. 3.25 – Interface de modification d'une ressource

3.3.5 Suppression d'une ressource

On appel la fonction de suppression via le menu : Edit -¿ Delete -¿ delete Resource

FIG. 3.26 – Interface de modification d'une ressource

Conclusion

Ce stage nous aura permis de découvrir le language ADA ainsi que GtkAda. Avoir apris GtkAda nous permettra de pouvoir utiliser Gtk avec de tres nombreux language étant données l'uniformité de ce toolkit. Nous avons également pu découvrir les algorithmes d'ordonnancements tels que RM et DM, les notions de précédences, de ressources partagées et de protocol de partage comme PCP, PIP.

Ce travail en groupe a également été trés enrichissant particuliérement pour l'utilisation cd CVS , mal connu jusqu'alors.

Globalement, ce stage c'est bien déroulé, nous avons pu rencontrer de nombreux problemes que nous avons pu résoudre par une recherche personnel de documentation la plupart du temps et a l'aide de Mr Singhoff autrement.

Quelques améliorations sont encore a aporter a cheddar, mais par manque de temps nous n'avons pas pu les réaliser

Annexe A

Arbres de test

A.1 Opération sur les tâches

Voici l'arbre de test des opérations sur les tâches. Il répertorie tous les tests de fonctionnement effectués et leur résultats. Les tests de suppression de tâches ont montrés que les dépendences de celle-ci sont bien supprimées. Il apparaît quelque fois un bug sur l'update de période et Jitter mais sa cause reste inexpliqué. Il semblerais que cela soit du à une certaine utilisation de cheddar mais celle-ci n'as pas pu être précisée.

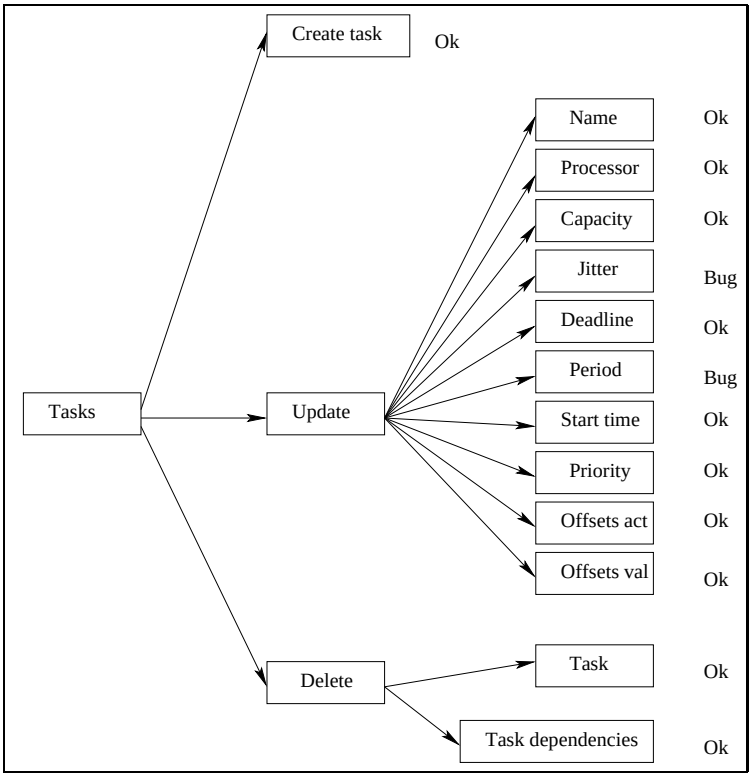


FIG. A.1 – Arbre de test des tâches

A.1.1 Validité des paramètres

Voici l'arbre de test de validité des paramètres. Il répertorie tous les tests de fonctionnement effectués et leur résultats. Le contrôle de non négativité est Ok.

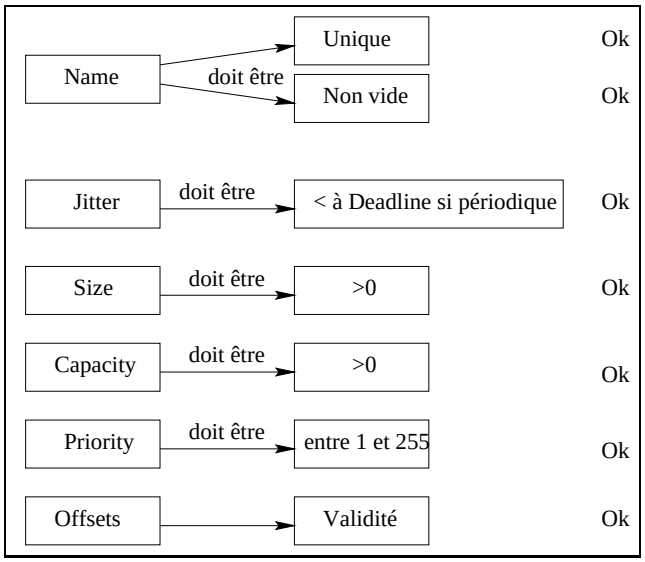


FIG. A.2 – Arbre de test des paramètres

A.2 Opération sur les Messages

Voici l'arbre de test des opérations sur les messages. Il répertorie tous les tests de fonctionnement effectués et leur résultats. Les tests de suppression de messages ont montrés que les dépendences de ceux-ci sont bien supprimées.

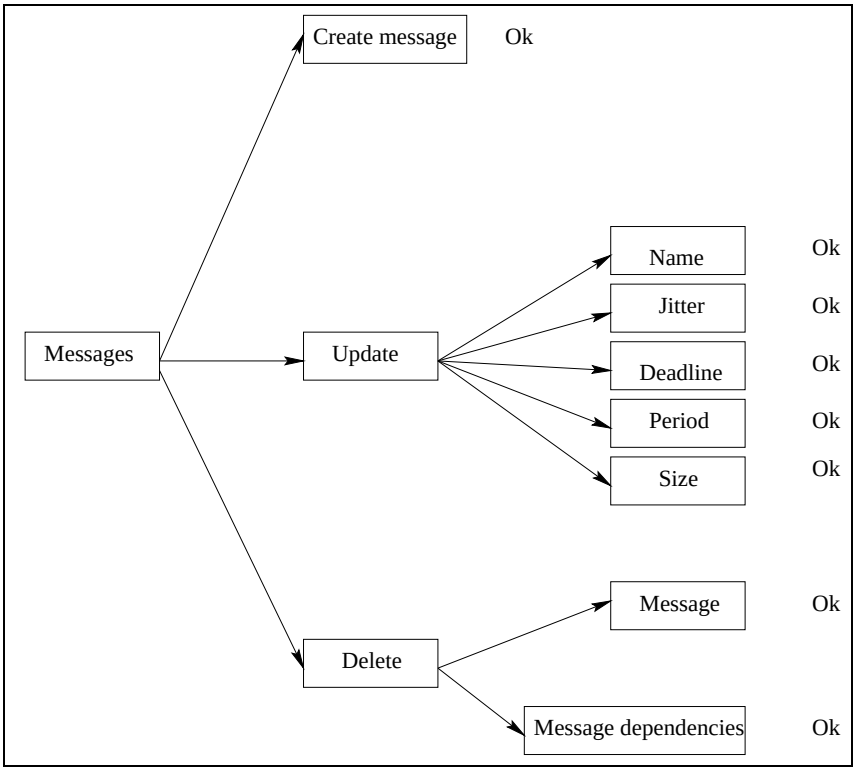


FIG. A.3 – Arbre de test des messages

A.2.1 Validité des paramètres

Voici l'arbre de test de validité des paramètres. Il répertorie tous les tests de fonctionnement effectués et leur résultats. Le contrôle de non négativité est Ok.

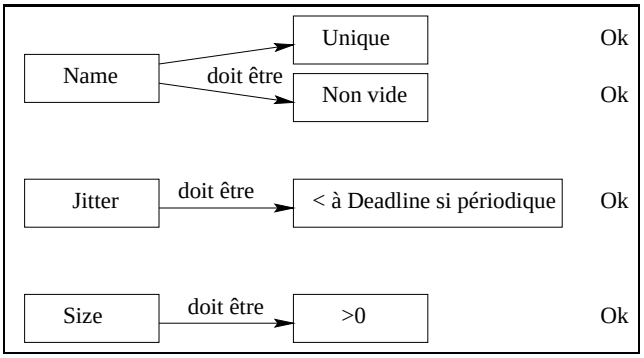


FIG. A.4 – Arbre de test des paramètres